MPS223 - Mathematics and Statistics in Action

Topic 1: Statistical analysis of climate-based data

2026-02-03

Contents

About these notes 5

1 Set up your R environment 7

2 Project overview 9

I Data processing 11

3 Importing data 13
3.1 Importing a .csv file 13
3.2 Importing Excel .xlsx files 14
3.3 Importing a plain text file 14
3.4 Importing online data 15

4 Strings 17
4.1 Example: (fictitous) exam mark data 17
4.2 Importing text data with `read_lines()` 17
4.3 Finding (and replacing) characters in strings 18
4.4 Subsetting strings 20
4.5 Making a data frame 20
4.6 Exercise 20
4.7 Further reading 21

5 Manipulating data frames - recap 23
5.1 Chaining commands together with the pipe operator `%>%` 24
5.2 Ordering the rows by a variable with the `arrange()` command 25
5.3 Selecting rows with the `filter()` command 25
5.4 Viewing and extracting data from a column 26
5.5 Creating new columns in a data frame with the `mutate()` command 27

6 Summary statistics - recap 29
6.1 Calculating summary statistics with the `summary()` command 29
6.2 Calculating individual summary statistics 30
6.3 Computing summaries per group 30
6.4 Exercise 31

7 Missing data 33
7.1 `NA` 33
7.2 Functions and `NA` 33

II Plotting data **35**

8 Making plots with `ggplot2` - recap **37**
8.1 The general syntax 38
8.2 Scatter plots 39
8.3 Histograms 43
8.4 Box plots 45
8.5 'Global' aesthetics 48
8.6 Facets 49
8.7 Exercises 50
8.8 Further reading 50

9 Presentation of plots **53**
9.1 The basics 53
9.2 The caption test 53
9.3 Caption or title? 54
9.4 Customising the appearance of a plot 56
9.5 Refining a plot: an example 56
9.6 Exercises 60
9.7 Data sources 61

10 Maps with `leaflet` **65**
10.1 A basic map 65
10.2 Indicating points on a map 67
10.3 Exercise 68
10.4 Further reading 68

III Statistical analysis **69**

11 Computing using loops **71**
11.1 Repeating a process with a `for` loop 71
11.2 Example: cleaning two text files 71
11.3 Example: averaging over groups 73
11.4 Exercise 75

12 Relationships in data **77**
12.1 Covariance and correlation - recap 78
12.2 Simple linear regression 81
12.3 Exploring variation over time 88
12.4 Exercise 93

13 Imputing missing data **95**
13.1 Visualising missing data 97
13.2 Exercise 98

About these notes

These notes support the work on MPS223 Topic 1: *Statistical analysis of climate-based data*, and build on the R notes from MAS109: Probability and Data Science (Introduction to R; Exploratory Data Analysis Using R) to provide information on additional exploratory data analysis (EDA) and statistical methods required for this work.

Topics covered include:

- Data processing for data provided as text;
- A recap on EDA approaches: Manipulating, summarising and plotting data;
- Statistical inference: computing using ‘for loops’, exploring relationships and patterns in data and handling missing values.

Chapter 1

Set up your R environment

Before starting your analysis, it is important to set up your working environment in R. When working in a group, it is useful to have the same base set-up so that you can move code between group members and the code will run easily.

Tasks

In the first Session, we will implement the following steps together to set up your R environment - **You will only need to do this once**:

1. Using Windows Explorer, create a folder for this module on your university 'U:' drive with the filepath `U:/MPS223/`

2. In your new MPS223 folder, create another new folder called `Project-1_Climate`

3. Set up an `R-Project` to contain your work:

- In R-studio, click on the `project` button at the top right of the screen (if you don't have a project open, the button will say 'Project: (None)' on it), and select `New Project...`

- On the dialogue box that appears, click on `Existing Directory` and then use the `Browse...` option to locate your new folder at: `U:/MPS223/Project-1_Climate`. Then click on `Create Project` to create an R-Project in your folder to hold your work.

This process will create an object called `Project-1_Climate.Rproj` in your `Project-1_Climate` folder. By creating the project, we set the working directory for R to use, and our files become easily accessible in R-studio.

4. In your `Project-1_Climate` folder, create another new folder called `data`. We will use this folder to store any data files required for the exercises in these notes, as well as the data for your group project.

5. On the module Blackboard page, go to the `Computer Labs - Examples data .zip file` in the Learning Materials section for Project 1. Download the `MPS223_Project1_data.zip` file containing all of the data files required for the notes and exercises, and unpack this file so that all of the data files are saved directly in your `data` folder.

6. Finally, you will need to install the `tidyverse` set of R packages into R. The Tidyverse is a collection of R packages designed for data science. We will be using a number of these packages

(in particular, a package for data manipulation called dplyr, a package for editing strings called stringr and a package for graphics called ggplot2) in this course. Run the following command in your R-console to install it:

```r
install.packages("tidyverse")
```

You only need to install this package once, but **every session**, you will need to load it with the command

```r
library(tidyverse)
```

Chapter 2

Project overview

Brief

This group project is orientated around climate data. Specifically, the cleaning of data, the analysis of data using appropriate exploratory data analysis (EDA) methods, and presenting it to an audience in a way which is a bit more modern than the traditional "stand in front of an audience" method.

Objectives

Our aim for this project is to simulate a data analysis task which a graduate might be asked to undertake in a workplace. By the end of this project, students should be able to:

1. Conduct themselves professionally and efficiently working in small groups.
2. Clean (i.e. appropriately format) a data set and analyse it using appropriate methods.
3. Work in an organised manner and to a tight deadline.
4. Present your findings as a group via a recorded presentation (i.e. you will **not** present to a live audience.)

Project

Scope

Your group project is to explore, analyse and interpret a data set collected from the Met Office website and present your work / findings in the form of a video presentation. Your grade for this assignment will be marked based on your video presentation and project log file, as a group.

Specifics

We do not expect you to analyse in full, the whole data set as this would be a lot of work. We do expect you to discuss the data set in full as part of an introduction, then perhaps focus on one or two attributes for the majority of your presentation. For example, you might pick two of the weather variables, such as maximum temperature and rainfall and compare / discuss / analyse those in detail.

Part I

Data processing

Chapter 3

Importing data

In this section we summarise R commands for importing data from .csv files and Excel spreadsheets, from plain text files, and directly from the web.

First, let's load the `tidyverse` library, if you haven't already.

```
library(tidyverse)
```

3.1 Importing a .csv file

A common format for data is the "comma separated variables" (csv) format. We can use the command `read_csv()`, with the path to the target file in quotes (if you can, use the path relative to your project directory), to read data of this format into R. This command is from the readr package (Wickham et al., 2018). As an example, we'll import the `RIOolympics.csv` file and store it as a data frame called `medals`.

```
medals <- read_csv("data/RIOolympics.csv")
```

```
## Rows: 87 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): code, country
## dbl (4): gold, silver, bronze, total
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

To view the first few rows:

```
head(medals)
```

```
## # A tibble: 6 x 6
##   code  country               gold silver bronze total
##   <chr> <chr>                <dbl>  <dbl>  <dbl> <dbl>
## 1 USA   UNITED STATES           46     37     38   121
## 2 GBR   GREAT BRITAIN           27     23     17    67
## 3 CHN   CHINA                   26     18     26    70
## 4 RUS   RUSSIAN FEDERATION      19     18     19    56
## 5 DEU    GERMANY                     17       10       15     42
## 6 JPN    JAPAN                       12        8       21     41
```

Additional arguments

Use `?read_csv` to see the full list of arguments. Two useful ones are

- `skip`: use this to skip lines at the start of the `.csv` file, for example if there's some text that you need to ignore;
- `col_names`: you will normally use the default value `TRUE`, but you can specify a character vector of column names if you want/need to.

3.2 Importing Excel .xlsx files

If your data is an Excel spreadsheet in `.xlsx` format, you can either save it in Excel as a `.csv` file, or you can use the `read_xlsx()` command (for which you will need to load the `readxl` package first). For example, to import a file `spreadsheet.xlsx`, you would use the command

```
library(readxl)
mydata <- read_xlsx("spreadsheet.xlsx")
```

Here, use `?read_xlsx` to see the full list of arguments.

3.3 Importing a plain text file

We can import plain text files (typically files with a `.txt` extension) with the command `read_table()`. For example, we have a data file called `flintsdata.txt`. Although the file is a plain text file, the data appear formatted as a two-column table, with headers `breadth` and `length`. We import the data (and create a new data frame called `flints`) as follows.

```
flints <- read_table("data/flintsdata.txt")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   breadth = col_double(),
##   length = col_double()
## )
```

```
head(flints)
```

```
## # A tibble: 6 x 2
##   breadth length
##     <dbl>  <dbl>
## 1    1.97   4.37
## 2    2.44   5.43
## 3    2.22   5
## 4    2.1    4.38
## 5    2.43   5.45
## 6    2.5    5.8
```

As with the other functions for importing data, there are useful arguments for skipping lines, defining column names and so on. See `?read_table` for details.

3.4 Importing online data

If you are online, R can read in data directly from the web: you just give the full web address as the file name, e.g.

```
maths <- read_csv("https://oakleyj.github.io/exampledata/maths.csv")
```

This can be convenient, though it may slow down your code if the file is large, and there is a risk that your code won't run in the future: you can't be sure that the file will always be there!

Nevertheless, downloading a data file via R (or another language) is preferable to clicking buttons/links on a website, as you will have a record of where you got the data from. We recommend that you download your data with R, but then save a copy on your computer, and that your analysis uses your copied version.

We can export data from R to a external file, e.g. csv format with the `write_csv()` command, e.g.

```
write_csv(maths, file = "data/myCopyOfMaths.csv")
```

We can also use the `download.file()` command, which downloads and saves the file, but doesn't keep the data within R:

```
download.file(url = "https://oakleyj.github.io/exampledata/maths.csv",
              destfile = "data/myCopyOfMaths.csv")
```

> When working with someone else's data, the data almost certainly won't be in the format that you need; you'll need to do some editing/re-arranging of the data. If possible, do not make any edits to the original data file: do any editing you need inside R. That way, you will have a record of how you got from the original data file to the data you actually worked with in your analysis.

Chapter 4

Strings

Here, we'll look at working with text and strings, which can be more awkward to deal with compared with purely numerical data. In this chapter, as an example, we'll consider a data set provided as a plain text file, in which there are some formatting problems we'll need to deal with.

4.1 Example: (fictitous) exam mark data

We'll consider a small text file of fictitious data. The file is called `stat101.txt`, and looks like this.

```
STAT101 module marks
29/06/20

student cwk exam
12015    55   62
12468    78   84
11560    55   40*
12589    62   --

* denotes resit attempt - capped at 40
```

Some problems with working with this data in R would be

- awkward labels attached to numbers (e.g. `40*`);
- handling of missing data: here `--` has been used, whereas R uses `NA`;
- lines of text both before and after the data.

> It may be tempting to edit a file such as this by hand in a text editor. Try not to do this! It may not be practical if the file is large, and it's hard to keep a good record of what edits you made.

4.2 Importing text data with `read_lines()`

If the text file only contained a table (and so looked like a data frame), we could use `readr::read_table()` to import it. But if we need to do any processing of the text first, a better option may be to first import the file with `readr::read_lines()`. This will create a vector of character strings, where each element is one whole line of the text file.

```
read_lines("data/stat101.txt")
```

```
##  [1] "STAT101 module marks"
##  [2] "29/06/20"
##  [3] ""
##  [4] "student cwk exam"
##  [5] "12015    55   62"
##  [6] "12468    78   84"
##  [7] "11560    55   40* "
##  [8] "12589    62   -- "
##  [9] ""
## [10] "* denotes resit attempt - capped at 40"
```

- as with other commands for importing data, `read_lines()` can import files directly from websites - just give the full url;
- we can use the argument `skip` to skip lines at the start (we might skip lines 1-3 here, but I will leave them in for now);
- the argument `skip_empty_rows` is `FALSE` by default, but I will set it to `TRUE` to skip rows 3 and 9:

```
examTextRaw <- read_lines("data/stat101.txt",
                          skip_empty_rows = TRUE)
```

4.3 Finding (and replacing) characters in strings

We can test for equality of entire strings in the usual way, e.g.

```
x <- c("red house", "blue car")
x == "red house"
```

```
## [1]  TRUE FALSE
```

but here, we will want to search *within* a string for some text, e.g., how to determine which elements of `x` contain the word `red`? Here, we will make use of the `stringr` package (Wickham, 2019).

4.3.1 Finding text with `str_which()`

Suppose we want to find the line with the column headings. Here, will do this by finding which line contains the text `student` (assuming we know that's what we need to look for):

```
str_which(examTextRaw, "student")
```

```
## [1] 3
```

4.3.2 Escape characters and regular expressions

Some characters have special meaning, which makes it harder to search for them. If we wanted to find lines containing `*`, this won't work:

```
str_which(examTextRaw, "*")
```

Here, we have to insert two backslash symbols, so that R understands we are searching for a `*`:

```
str_which(examTextRaw, "\\*")
```

```
## [1] 6 8
```

To do more complicated searches, one can make use of **regular expressions**. Regular expressions describe particular patterns of text, and can be used in many different programming languages. We won't cover these here, but further reading is given at the of this chapter.

4.3.3 Replacing or removing text

Suppose we want to replace `--` by `NA`, to indicate a missing value. We do

```
str_replace_all(examTextRaw, pattern = "--",
                replacement = "NA")
```

```
## [1] "STAT101 module marks"
## [2] "29/06/20"
## [3] "student cwk exam"
## [4] "12015    55   62"
## [5] "12468    78   84"
## [6] "11560    55   40* "
## [7] "12589    62   NA "
## [8] "* denotes resit attempt - capped at 40"
```

To delete text, we can either set `replacement = ""` in the above or use `str_remove_all()`. For example, to get rid of the asterisks, we would do

```
str_remove_all(examTextRaw, pattern = "\\*")
```

```
## [1] "STAT101 module marks"
## [2] "29/06/20"
## [3] "student cwk exam"
## [4] "12015    55   62"
## [5] "12468    78   84"
## [6] "11560    55   40 "
## [7] "12589    62   -- "
## [8] " denotes resit attempt - capped at 40"
```

4.3.4 Removing white space at start/end of strings

Blank spaces at the end of a string can cause problems, as R might think there is an extra column of data. We can get rid of these with `str_trim()`

```
str_trim(examTextRaw)
```

```
## [1] "STAT101 module marks"
## [2] "29/06/20"
## [3] "student cwk exam"
## [4] "12015    55   62"
## [5] "12468    78   84"
## [6] "11560    55   40*"
## [7] "12589    62   --"
## [8] "* denotes resit attempt - capped at 40"
```

4.4 Subsetting strings

We might want to search for some text at a particular place in a string, or just extract part of a string. We can do this with `str_sub()`. For example, to extract the module code from the data, we could do

```r
str_sub(examTextRaw[1], start = 1, end = 7)
```

```
## [1] "STAT101"
```

4.5 Making a data frame

We'll first do some operations to clean up the text:

```r
examTextClean <- examTextRaw %>%
  str_remove_all(pattern = "\\*") %>%
  str_replace_all(pattern = "--",
                  replacement = "NA") %>%
  str_trim()
```

and then do some searching to see which lines we want to use in our data frame:

```r
header <- str_which(examTextClean, pattern = "student")
endLine <- str_which(examTextClean, pattern = "denotes")
```

Then, we can make a data frame with

```r
read_table(examTextClean[header:(endLine - 1)])
```

```
## # A tibble: 4 x 3
##   student   cwk  exam
##     <dbl> <dbl> <dbl>
## 1   12015    55    62
## 2   12468    78    84
## 3   11560    55    40
## 4   12589    62    NA
```

4.6 Exercise

Exercise 4.1. The file `FarmYield.txt` contains (fictitous) data on crop yields for farms in an area of the South West region of the UK.

For this data file, write some R code to import and clean the data, and then convert the cleaned data into a usable data frame.

Use the `mean()` function to compute the average yield of Oilseed rape (in tonnes per hectare) for the farms in this area.

4.7 Further reading

- For more on `stringr`, see Chapter 14 of R for Data Science
- In particular, see Section 14.3 on regular expressions

Chapter 5

Manipulating data frames - recap

Once we have our data in a single data frame, we might wish to obtain some simple summary statistics, perhaps within subgroups of the data, and/or understand the structure of the data better. The package `dplyr` (Wickham et al., 2020b) has various functions for working with data frames, and we will recap the key ones here.

(Artwork by @allison_horst)

To illustrate the commands, we will use the data `maths.csv`, which can be read into R using:

```
maths <- read_csv("https://oakleyj.github.io/exampledata/maths.csv")
```

This is a data set of maths scores from the PISA 2015 maths tests, with data obtained from OECD and the World Bank. In addition to country and continent, it contains the following columns:

1. `score`: the mean mathematics score in the 2015 PISA test;
2. `gdp`: the gross domestic product per capita (GDP divided by the estimated population size), measured in US$;
3. `gini`: the Gini coefficient (as a percentage). This is an estimate of income inequality, with larger

values indicating more income inequality;

4. `homework`: an estimate of the average number of hours per week spent on homework by 15 year-olds, from a survey in 2012;
5. `start.age`: the age (in years) in which children start school.

The data are arranged in the data frame as they are in the underlying .csv file, with one row per country, and one column per variable. Typing `maths` (and pressing return) in the R console will display the first ten rows only, and as many columns as will fit in the window.

```r
maths
```

```
## # A tibble: 70 x 7
##    country         continent     score   gdp  gini homework start.age
##    <chr>           <chr>         <dbl> <dbl> <dbl>    <dbl>     <dbl>
##  1 Albania         Europe          413  4147  29        5.1         6
##  2 Algeria         Africa          360  3844  27.6     NA           6
##  3 Argentina       South America   409 12449  42.7      3.7         6
##  4 Australia       Oceanea         494 49928  34.7      6           5
##  5 Austria         Europe          497 44177  30.5      4.5         6
##  6 B-S-J-G (China) Asia            531  8123  42.2     13.8         6
##  7 Belgium         Europe          507 41096  28.1      5.5         6
##  8 Brazil          South America   377  8650  51.3      3.3         6
##  9 Bulgaria        Europe          441  7351  37.4      5.6         7
## 10 Canada          North America   516 42158  34        5.5         6
## # i 60 more rows
```

Within the data frame, we see that the column names are `country`, `continent`, `score`, `gdp`, `gini`, `homework` and `start.age`, which we will use in various commands described shortly.

If we want to see all 70 rows, we can either use the command

```r
print(maths, n = 70)
```

or, in RStudio, we can click on `maths` in the Environment window.

5.1 Chaining commands together with the pipe operator %>%

We'll first look at the 'pipe operator' `%>%`. (It's a matter of personal preference whether you use it in your code or not, but you may find it makes your code easier to read.)

The pipe operator `%>%` takes whatever the output is from the left hand side, and uses it as the first argument in the function on the next line. In general,

```r
myfunction(x, y)
```

can be written using the pipe operator as

```r
x %>%
  myfunction(y)
```

5.2 Ordering the rows by a variable with the arrange() command

Suppose we want to see which countries got the highest score: we want to arrange the rows in the dataframe maths in order according to the values in the column score. To do this we use the command

```
maths %>%
  arrange(score)
```

```
## # A tibble: 70 x 7
##    country            continent     score   gdp  gini homework start.age
##    <chr>              <chr>         <dbl> <dbl> <dbl>    <dbl>     <dbl>
##  1 Dominican Republic North America   328  6722  44.9     NA           6
##  2 Algeria            Africa          360  3844  27.6     NA           6
##  3 Tunisia            Africa          367  3689  35.8      3.5         6
##  4 Macedonia, FYR     Europe          371  5237  35.6     NA           6
##  5 Brazil             South America   377  8650  51.3      3.3         6
##  6 Jordan             Asia            380  4088  33.7      4.2         6
##  7 Indonesia          Asia            386  3570  39.5      4.9         7
##  8 Peru               South America   387  6046  44.3      5.5         6
##  9 Colombia           South America   390  5806  51.1      5.3         6
## 10 Lebanon            Asia            396  7914  31.8      3.3         6
## # i 60 more rows
```

This has arranged the rows in ascending order of score. To see them in descending order, we include the desc() command:

```
maths %>%
  arrange(desc(score))
```

```
## # A tibble: 70 x 7
##    country              continent     score   gdp  gini homework start.age
##    <chr>                <chr>         <dbl> <dbl> <dbl>    <dbl>     <dbl>
##  1 Singapore            Asia            564 52961 NA          9.4          6
##  2 Hong Kong SAR, China Asia            548 43681 NA          6            6
##  3 Macao SAR, China     Asia            544 73187 NA          5.9          6
##  4 Japan                Asia            532 38894 32.1        3.8          6
##  5 B-S-J-G (China)      Asia            531  8123 42.2       13.8          6
##  6 Korea, Rep.          Asia            524 27539 31.6        2.9          6
##  7 Switzerland          Europe          521 78813 32.5        4            7
##  8 Estonia              Europe          520 17575 34.6        6.9          7
##  9 Canada               North America   516 42158 34          5.5          6
## 10 Netherlands          Europe          512 45295 28.6        5.8          6
## # i 60 more rows
```

5.3 Selecting rows with the filter() command

If we want to view a subset of the rows, we can use the filter() command. For example, if we want the rows in the data frame maths where start.age takes the value 5 (i.e. children start school at age 5), we can do

```
maths %>%
  filter(start.age == 5)
```

```
## # A tibble: 6 x 7
##   country             continent     score   gdp  gini homework start.age
##   <chr>               <chr>         <dbl> <dbl> <dbl>    <dbl>     <dbl>
## 1 Australia           Oceanea         494 49928  34.7      6           5
## 2 Ireland             Europe          504 61606  31.9      7.3         5
## 3 Malta               Europe          479 25058  NA       NA           5
## 4 New Zealand         Oceanea         495 39427  NA        4.2         5
## 5 Trinidad and Tobago North America   417 15377  40.3     NA           5
## 6 United Kingdom      Europe          492 39899  34.1      4.9         5
```

Note the double equals sign `==`. This is used to test whether the left and right hand sides are equal: each country is included if its corresponding `start.age` is equal to 5. The UK is included above, but we'll give an example of selecting it anyway:

```
maths %>%
  filter(country == "United Kingdom")
```

```
## # A tibble: 1 x 7
##   country        continent score   gdp  gini homework start.age
##   <chr>          <chr>     <dbl> <dbl> <dbl>    <dbl>     <dbl>
## 1 United Kingdom Europe      492 39899  34.1      4.9         5
```

5.4 Viewing and extracting data from a column

For larger data frames (with many columns), we may wish to view a subset only. For example, to select the `score` and `country` columns only from the `maths` data frame, we do

```
maths %>%
  select(score, country)
```

```
## # A tibble: 70 x 2
##    score country
##    <dbl> <chr>
##  1   413 Albania
##  2   360 Algeria
##  3   409 Argentina
##  4   494 Australia
##  5   497 Austria
##  6   531 B-S-J-G (China)
##  7   507 Belgium
##  8   377 Brazil
##  9   441 Bulgaria
## 10   516 Canada
## # i 60 more rows
```

If we want to extract the values from a column, we use the syntax `dataframe-name$column-name`. For example, to extract the column `score` from the data frame `maths`, we do

```
maths$score
```

```
##  [1] 413 360 409 494 497 531 507 377 441 516 423 390 400 464 437 492 511 328 520
## [20] 511 493 404 506 454 548 477 488 386 504 470 490 532 380 460 524 482 396 478
## [39] 486 544 371 446 479 408 420 418 512 495 502 387 504 492 402 444 494 564 475
## [58] 510 486 494 521 415 417 367 420 427 492 470 418 495
```

We could then, for example, calculate the average (mean) of all the scores:

```
mean(maths$score)
```

```
## [1] 460.9714
```

5.5 Creating new columns in a data frame with the `mutate()` command

About half the countries have a GDP per capita greater than $17000. If we try the command

```
maths$gdp > 17000
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
## [13] FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
## [25]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
## [37] FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
## [49]  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
## [61]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE
```

this creates a new vector, in which the ith element will be `TRUE` if the `gdp` value for country i is greater than 17000, and `FALSE` otherwise. We will add this vector to the data frame, under the column name `wealthiest`. The command to create the new column is

```
mutate(maths, wealthiest = maths$gdp > 17000)
```

but this doesn't store the result. To put the new column in the `maths` dataframe, we do

```
maths <- maths %>%
  mutate(wealthiest = maths$gdp > 17000)
```

You may now have too many columns to see in your console, so to check this has worked, we will do

```
maths %>%
  select(country, gdp, wealthiest)
```

```
## # A tibble: 70 x 3
##    country           gdp wealthiest
##    <chr>           <dbl> <lgl>
##  1 Albania          4147 FALSE
##  2 Algeria          3844 FALSE
##  3 Argentina       12449 FALSE
##  4 Australia       49928 TRUE
##  5 Austria         44177 TRUE
##  6 B-S-J-G (China)  8123 FALSE
##  7 Belgium         41096 TRUE
##  8 Brazil           8650 FALSE
##  9 Bulgaria         7351 FALSE
## 10 Canada          42158 TRUE
## # i 60 more rows
```

Chapter 6

Summary statistics - recap

6.1 Calculating summary statistics with the `summary()` command

We can use summary statistics to highlight key features of the data. For example, for our `maths` data set, we might want to compare the UK's score with that in other countries. Recall that `maths$score` extracts the maths scores for the 70 countries. We can obtain various summary statistics for this variable with the command

```
summary(maths$score)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   328.0   417.2   477.5   461.0   500.8   564.0
```

The output from the `summary()` command here tells us that:

- the smallest observed score was 328;
- the 25th percentile, or 0.25 quantile, is 417.2: approximately 25% of the countries have scores less than (or equal to) 417.2;
- the median score was 477.5;
- the arithmetic mean (sum of all the scores, divided by 70) for the 70 countries was 461;
- the 75th percentile, or 0.75 quantile, is 500.8: approximately 75% of the countries have scores less than (or equal to) 500.8.
- the largest observed score was 564.

The interquartile range [the difference between the 75th percentile (0.75 quantile) and 25th percentile (0.25 quantile)] can be obtained in R with the command

```
IQR(maths$score)
```

```
## [1] 83.5
```

We've seen that the UK's score was 492. This ranks the UK outside the top 25%, but inside the top 50%. We can find the actual rank as follows:

```
sum(maths$score > 492)
```

```
## [1] 25
```

This tells us that the UK ranked 26th: 25 countries got higher scores.

6.2 Calculating individual summary statistics

Individually, we could have calculated these summaries as follows:

```
min(maths$score)
quantile(maths$score, 0.25)
quantile(maths$score, 0.5)
mean(maths$score)
quantile(maths$score, 0.75)
max(maths$score)
```

This may be more convenient if we just want a particular summary statistic, and/or want to store the result for use later on.

Notice here that 'variance' and 'standard deviation' are not included in the output from the `summary()` command. The commands to obtain these statistics are:

```
var(maths$score)
sd(maths$score)
```

And, if we wanted another quantile/percentile, say, the 90th percentile (0.9 quantile), we can do

```
quantile(maths$score, 0.9)
```

```
##   90%
## 520.1
```

This tells us that, approximately, 90% of the countries have scores less than or equal to 520.1.

6.3 Computing summaries per group

Suppose we want to know the mean score within particular groups, for example, continents. We can do this by chaining together the `group_by()` and `summarise()` commands.

```
maths %>%
  group_by(continent) %>%
  summarise(MeanScore = mean(score))
```

```
## # A tibble: 6 x 2
##   continent     MeanScore
##   <chr>             <dbl>
## 1 Africa             364.
## 2 Asia               471.
## 3 Europe             476.
## 4 North America      423.
## 5 Oceanea            494.
## 6 South America      401.
```

We read this command as: "Start with the **maths** data frame, organise into groups based on the **continent** column, then create a new variable called **MeanScore**, which is the mean of the **score** variable within each group."

6.4 Exercise

Exercise 6.1. For this exercise, you will need to load the 'Brexit' data file, `Brexit.csv`. This file contains data from the 2016 UK referendum on leaving the European Union, along with data on proportions (as percentages) of adults with educational qualifications at 'level 4' and above (qualifications above A-level). The data are given for local authorities within the UK.

1. Find the 10 areas (the `Area` column) with the highest percentages of remain votes.
2. Find the mean percentage of remain votes within each `Region`.
3. Create a new column called `majorityRemain`, which indicates (using `TRUE` or `FALSE`) whether or not the percentage of remain votes in each `Area` was above 50%. Then, find the mean percentage of adults with level 4 qualifications (or higher), separately for areas with a majority in favour of remaining, and with a majority in favour of leaving.

Chapter 7

Missing data

It is common to find missing values when provided with a data set. In this Section, we'll briefly discuss how R represents and handles missing data.

7.1 NA

R represents a missing observation with NA. For example, we can create a vector with missing elements as follows,

```
x <- c(2, 4, NA, 6, NA)
```

and we can test to see if there are missing elements with the function is.na():

```
is.na(x)
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE
```

> If possible, do not remove missing data at the data cleaning/processing stage. Rather, store the missing values as NA, so you have a record of them, and then use appropriate methods to handle missing values inside R.

7.2 Functions and NA

Some functions will, by default, return NA if the vector has any missing elements, e.g.

```
mean(x)
```

```
## [1] NA
```

however, there is usually an argument to specify whether to remove missing values, e.g.

```
mean(x, na.rm = TRUE)
```

```
## [1] 4
```

Alternatively, the function na.omit() can be used to first remove missing values:

```
mean(na.omit(x))
```

```
## [1] 4
```

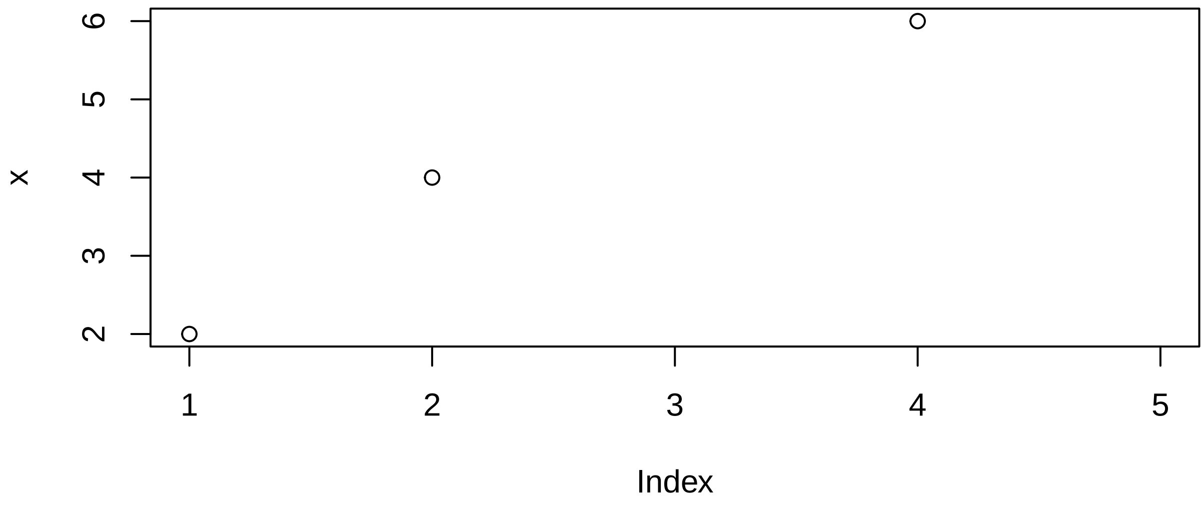
If used with a data frame, `na.omit()` will exclude rows where any single column has a missing value:

```
y <- 11:15
myData <- data.frame(x, y)
na.omit(myData)
```

```
##   x  y
## 1 2 11
## 2 4 12
## 4 6 14
```

Plot commands will typically ignore missing values (although you may get a warning message), for example

```
plot(x)
```



If we come across missing data, it's important to try to understand why the data are missing. For example, if in some clinical trial a patient drops out (resulting in missing data) because the treatment wasn't working, we can't just delete the patient from our data set for convenience; we'd be ignoring something important about how effective the treatment is.

Part II

Plotting data

Chapter 8

Making plots with ggplot2 - recap

R has different choices available for using plots. Two commonly used options are 'base graphics' (e.g. using commands such as `plot()`, `hist()`, `boxplot()`) and the package `ggplot2` (Wickham et al., 2020a), (Wickham, 2016). It's a matter of personal preference which you should use; I like to use both: base graphics for drawing diagrams, and `ggplot2` for plotting data from data frames. In these notes we will cover plotting with `ggplot2`.

(Artwork by @allison_horst)

In this section, we'll concentrate on how to make different types of plots. You'll almost certainly need to customise the appearance of your plot, and we'll cover that in the next section.

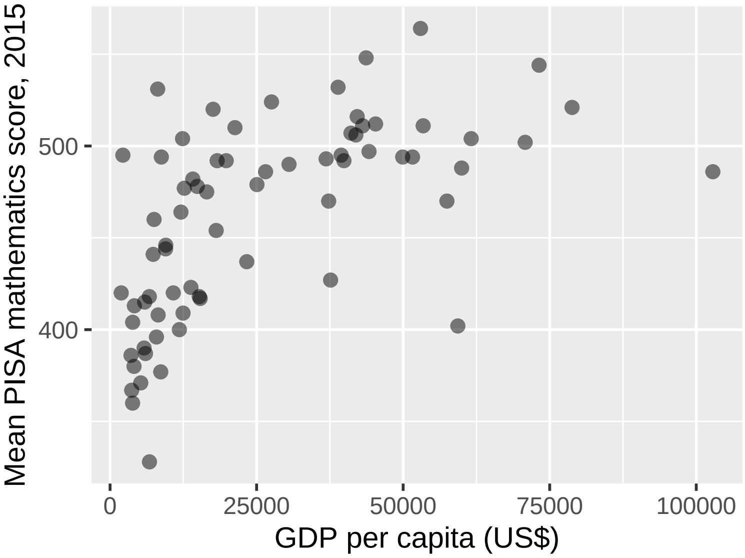
To illustrate the different types of plots, we will continue to use `maths` data example. To try these examples out on your own computer, you'll again need to load the `tidyverse` as well as import the `maths.csv` data and create the `wealthiest` column. The commands to do this are below.

```r
library(tidyverse)
maths <- read_csv("https://oakleyj.github.io/exampledata/maths.csv") %>%
  mutate(wealthiest = gdp > 17000)
```

8.1 The general syntax

As an example, we'll first produce a scatter plot:

```
ggplot(data = maths, aes(x = gdp, y = score)) +
  geom_point(size = 2, alpha = 0.5) +
  labs(x = "GDP per capita (US$)",
       y = "Mean PISA mathematics score, 2015")
```



The syntax can look a little complicated at first but you should get used to it! Some general points are as follows.

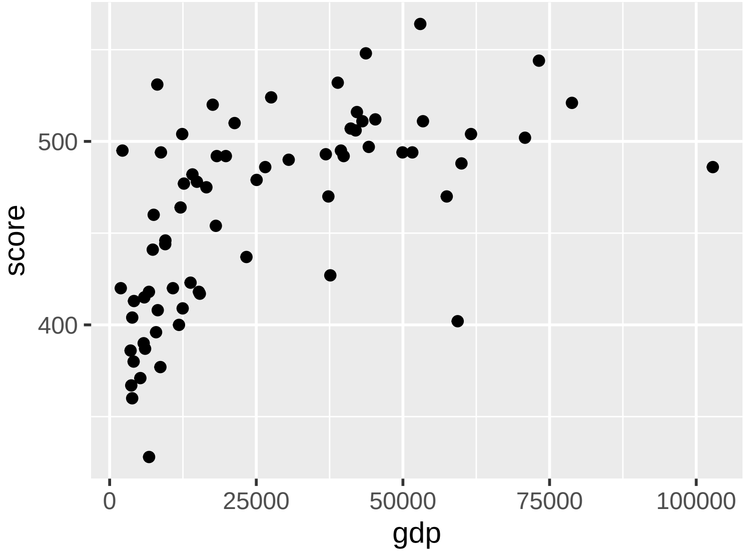
> There is quite a lot to take in here, so just skim read for now, and then have another look once you have seen some examples in the following sections.

- The data we want to plot must be in a data frame.
- All plots begin with a `ggplot()` command, specifying the data frame we are using (`maths` in this case).
- We use the `aes()` any time we want something on the plot to represent values in a data frame column, e.g
  - the position of a point on the x-axis or y-axis;
  - the colour of a point (different colours depending on variable values). This is sometimes referred to as 'mapping a column to an aesthetic'. In the example, we mapped the `gdp` column onto the x-axis, and the `score` column to the y-axis.
- Any feature of the points *not* related to columns in the data frame are specified *outside* of an `aes()` command. Here we used
  - `size` to make the points a little larger
  - `alpha` to make the points transparent (helpful when points are overlapping).
- The actual type of plot is specified as a "geom". There will be different `geom` commands for scatter plots, box plots, histograms etc. We used `geom_point()` to make a scatter plot.
- We use the `+` symbol to combine commands and make a single plot. In the example we've added a command to provide axes labels.

8.2 Scatter plots

We've seen a scatter plot already, but we'll repeat the basic syntax here:

```
ggplot(data = maths, aes(x = gdp, y = score)) +
  geom_point()
```



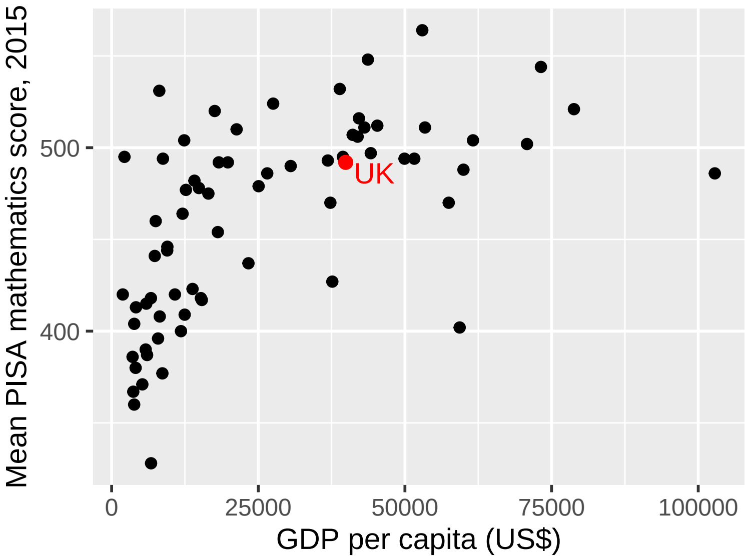
Note that, without the `labs()` command, the axes labels are just the column names.

> In any report that you write, you will almost certainly need to specify your own axes labels; the column names will not be suitable. In your labels, include the units of measurement, if appropriate.

8.2.1 Annotating a scatter plot

We may wish to annotate a plot, e.g. labelling a single observation. Here's an example of labelling the UK (GDP = 39899, score = 492). We'll also include proper axes labels.

```
ggplot(data = maths, aes(x = gdp, y = score)) +
  geom_point() +
  labs(x = "GDP per capita (US$)",
       y = "Mean PISA mathematics score, 2015") +
  annotate("point", x = 39899, y = 492,
           colour = "red", size = 2) +
  annotate("text", label = "UK", x = 39899,
           y = 492, colour = "red",
           hjust = -0.2, vjust = 1)
```

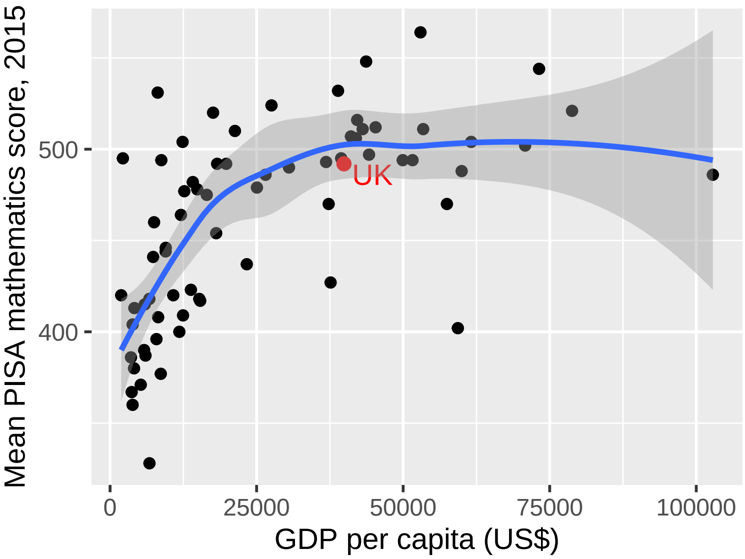


- The first `annotate()` command adds a red circle (`"point"`) at the coordinates $x = 39899$, $y = 492$ (corresponding to the UK), with the `size` set to 2 to make it a little larger.
- The second `annotate()` command adds some red text (UK) at the coordinates $x = 39899$, $y = 492$, with the arguments `hjust` and `vjust` shifting the text slightly horizontally and vertically, so that it appears next to rather than on top of the red dot. It can take a little trial and error to find the values for `hjust` and `vjust` that you are happy with.

8.2.2 Adding a trend

We can see clearly that the maths scores tend to increase as GDP per capita increases, but the relationship doesn't look linear. If we want to emphasise such a relationship, we can add the trend to the plot, using the extra line `geom_smooth()` in the plot command:

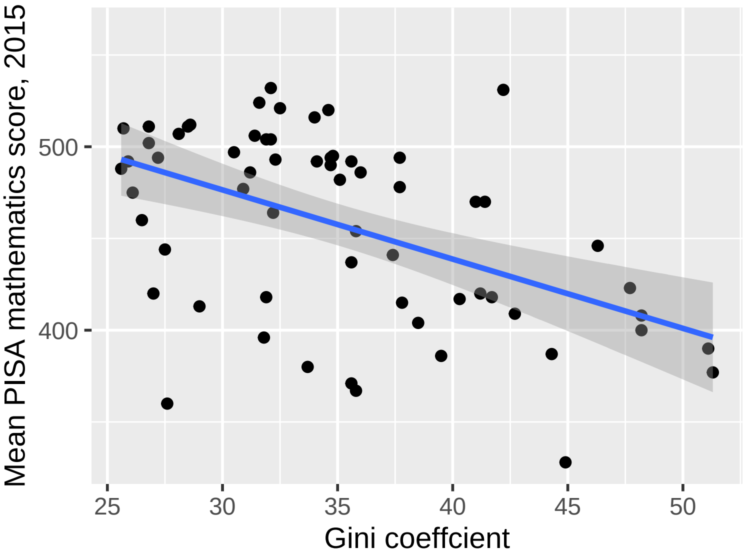
```
ggplot(data = maths, aes(x = gdp, y = score)) +
  geom_point() +
  labs(x = "GDP per capita (US$)",
       y = "Mean PISA mathematics score, 2015") +
  annotate("point", x = 39899,
           y = 492, colour = "red",
           size = 2) +
  annotate("text", label = "UK",
           x = 39899, y = 492,
           colour = "red",
           hjust = -0.2, vjust = 1) +
  geom_smooth()
```



The blue line shows the estimated trend. The grey shaded area indicates uncertainty about this trend (it's wider on the right hand side, because we have less data there).

If we want to plot a linear trend, we add the argument `method = "lm"` to `geom_smooth()` (we can use various regression methods within `geom_smooth()`):

```r
ggplot(data = maths,
       aes(x = gini, y = score)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Gini coeffcient",
       y = "Mean PISA mathematics score, 2015")
```



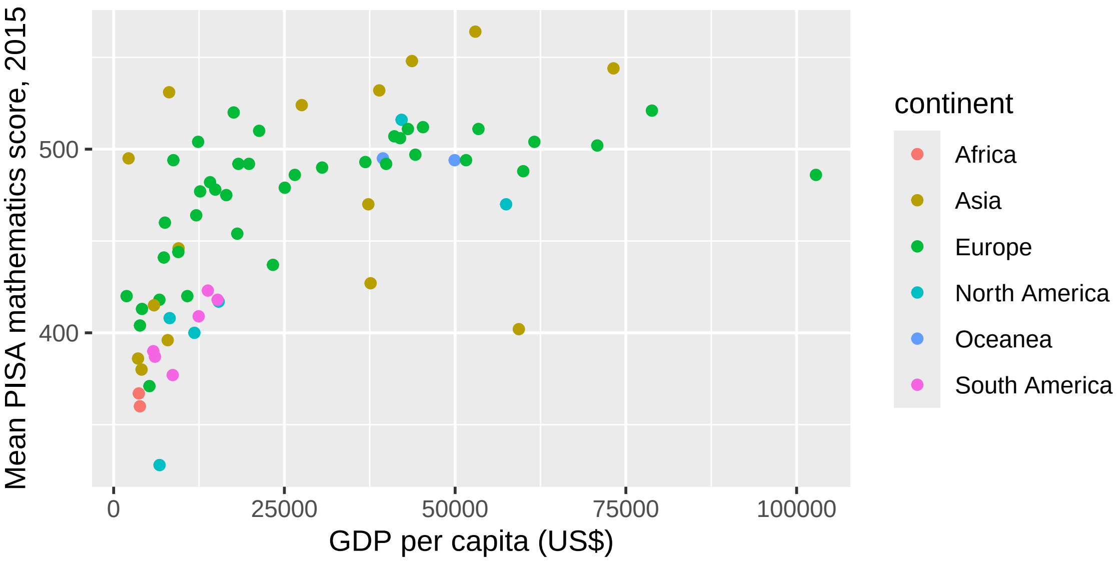
8.2.3 Representing more than two variables on a scatter plot

We can represent a third variable using colour. Additional variables can be presented using size and shape, although care is needed here; the plot could get difficult to read.

8.2.3.1 Colours for qualitative variables

Here's an example using colour to represent `continent`:

```
ggplot(data = maths,
       aes(x = gdp, y = score)) +
  geom_point(aes(colour = continent)) +
  labs(x = "GDP per capita (US$)",
       y = "Mean PISA mathematics score, 2015")
```

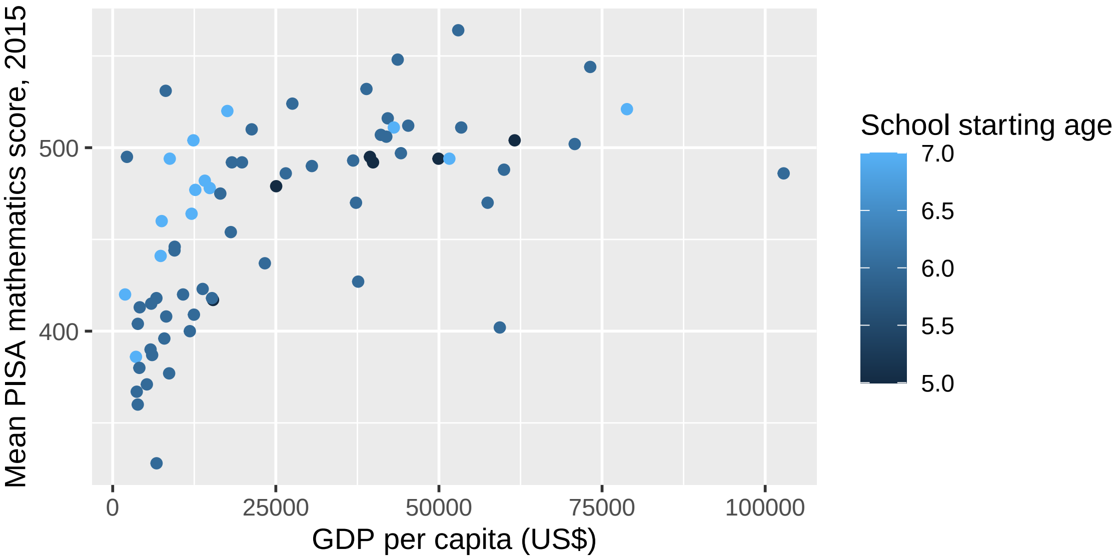


- Note that the mapping of `continent` to `colour` has to sit inside an `aes()` command.
- Colours can be specified manually. `ggplot2` will attempt to use distinctive colours by default, but this can be hard if there are too many groups.

8.2.3.2 Colours for quantitative variables

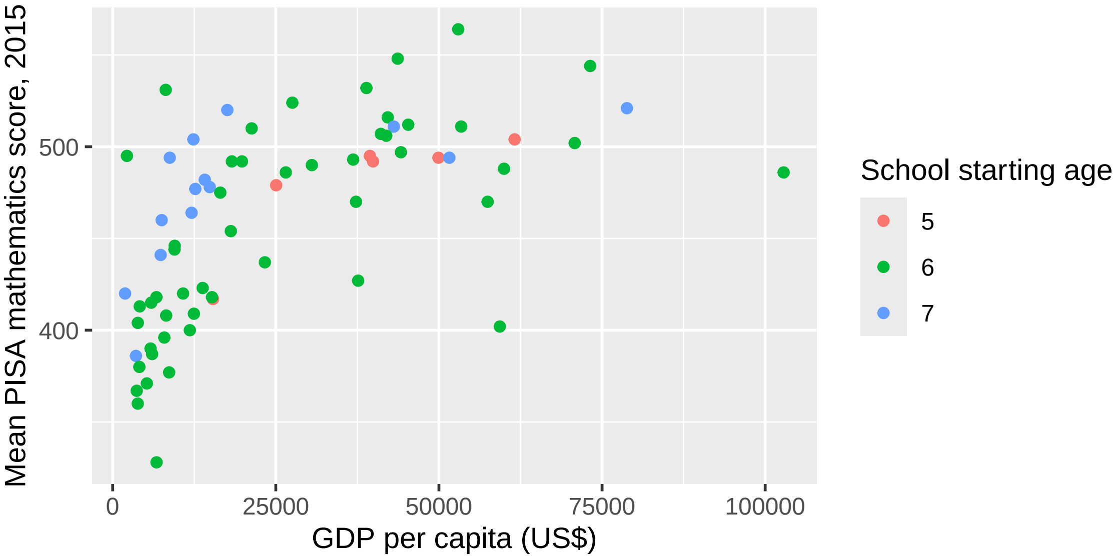
Here's an example using colour to represent `start.age`:

```
ggplot(data = maths,
       aes(x = gdp, y = score)) +
  geom_point(aes(colour = start.age)) +
  labs(x = "GDP per capita (US$)",
       y = "Mean PISA mathematics score, 2015",
       color = "School starting age" )
```



Note how a ‘continuous’ colour scale has been used. As there are only three distinct starting ages in the data, a ‘qualitative’ colour scale might be better here. To do this, we can convert `start.age` to a factor variable with the plot commands:

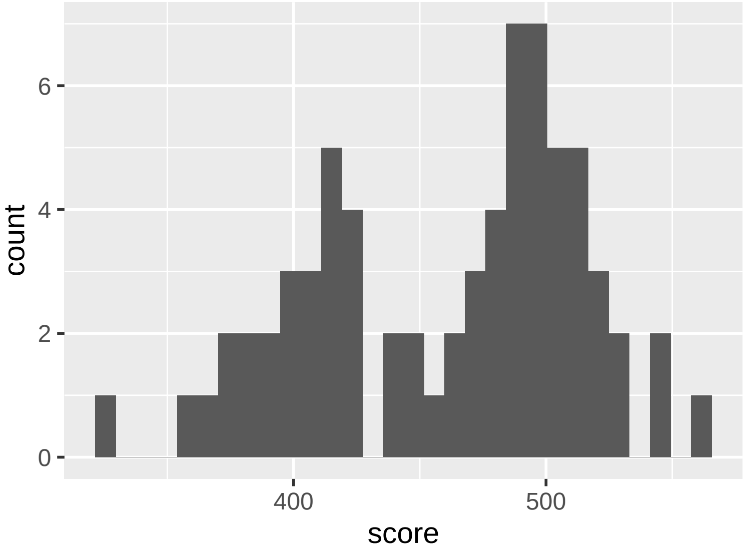
```
ggplot(data = maths,
       aes(x = gdp, y = score)) +
  geom_point(aes(colour = factor(start.age))) +
  labs(x = "GDP per capita (US$)",
       y = "Mean PISA mathematics score, 2015",
       color = "School starting age")
```



8.3 Histograms

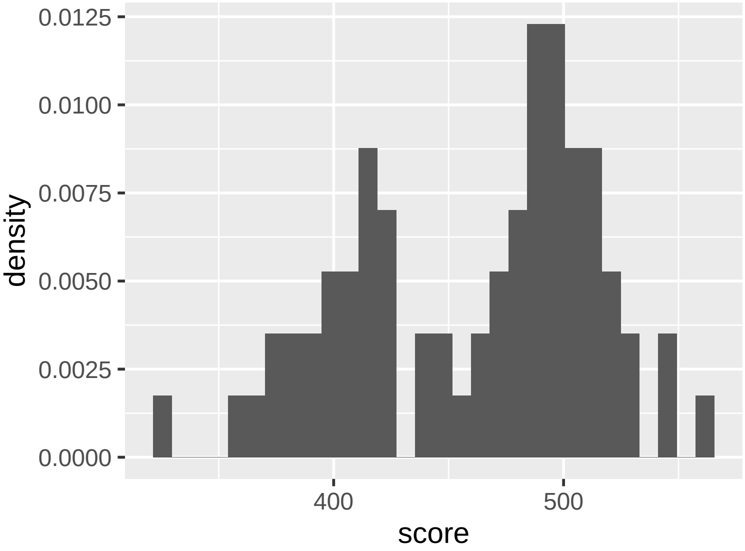
Here’s an example of a histogram of the `score` variable. We only map one column to an axis, and produce the histogram with `geom_histogram()`:

```
ggplot(data = maths, aes(x = score)) +
  geom_histogram()
```



We can scale the histogram so that the total area equals 1:

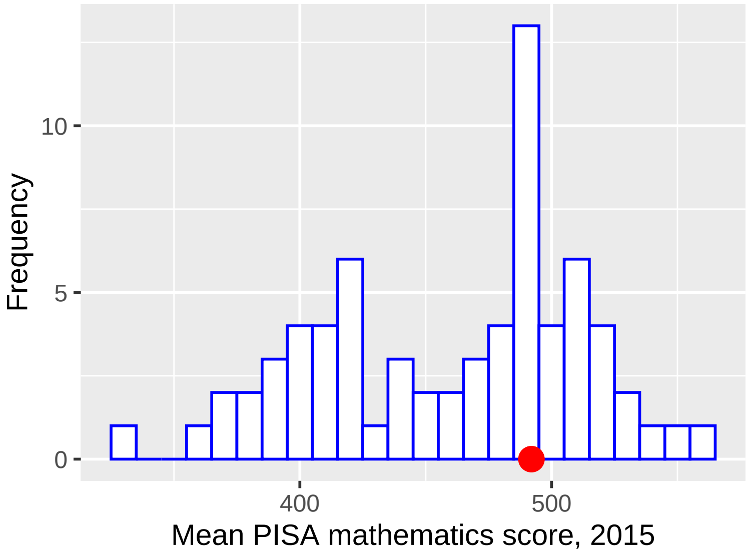
```
ggplot(data = maths, aes(x = score, y = ..density..)) +
  geom_histogram()
```



8.3.1 Customising a histogram plot in R

We'll redraw the plot with different colours, add a better axis label, specify a histogram bin-width of size 10, and indicate the UK's score with a red dot:

```
ggplot(data = maths, aes(x = score)) +
  geom_histogram(colour = "blue", fill = "white", binwidth = 10) +
  labs(x = "Mean PISA mathematics score, 2015", y = "Frequency") +
  annotate("point", x = 492, y = 0,  size = 4, colour = "red")
```

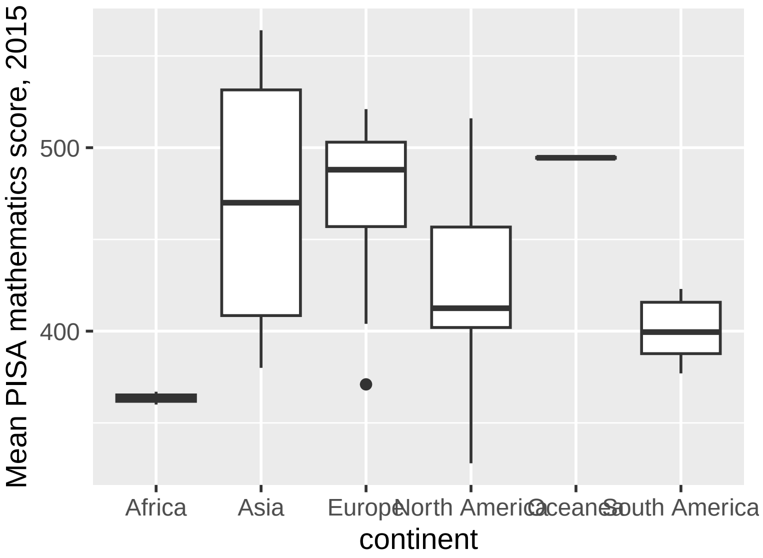


- The second line now includes extra arguments: `fill` sets the colour of the interior of the bars, and `colour` sets the colour of the bar edges. `binwidth` sets how wide each bar is on the x-axis;
- the third line (`labs`) specifies the label on the x-axis;
- the fourth line (`annotate`) draws a red circle at the coordinates $x = 492, y = 0$, and `size = 4` increases the size of the circle (the default value for `size` is 1).

8.4 Box plots

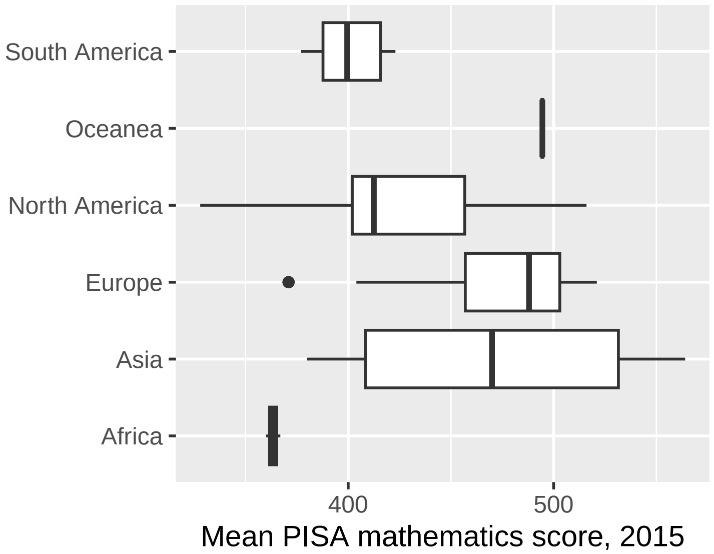
Box plots can be useful for comparing multiple distributions, although you may need to explain to your reader how to interpret them! Here's an example of a box plot to compare scores between the different continents:

```
ggplot(data = maths, aes(x = continent, y = score)) +
  geom_boxplot() +
  labs(y = "Mean PISA mathematics score, 2015")
```



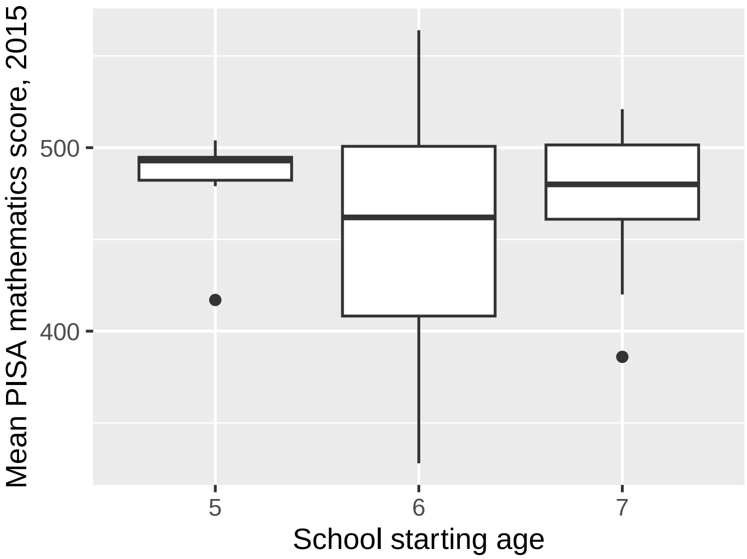
It can be difficult to fit all the labels in on the x-axis. A simple solution is to draw the box plot 'horizontally' (and we'll specify an empty y-axis label as we don't really need one).

```
ggplot(data = maths, aes(y = continent, x = score)) +
  geom_boxplot() +
  labs(x = "Mean PISA mathematics score, 2015",
       y = "")
```



If we wanted to do a box plot of `score` by `start.age`, we have to convert `start.age` to a factor variable using `as.factor()`

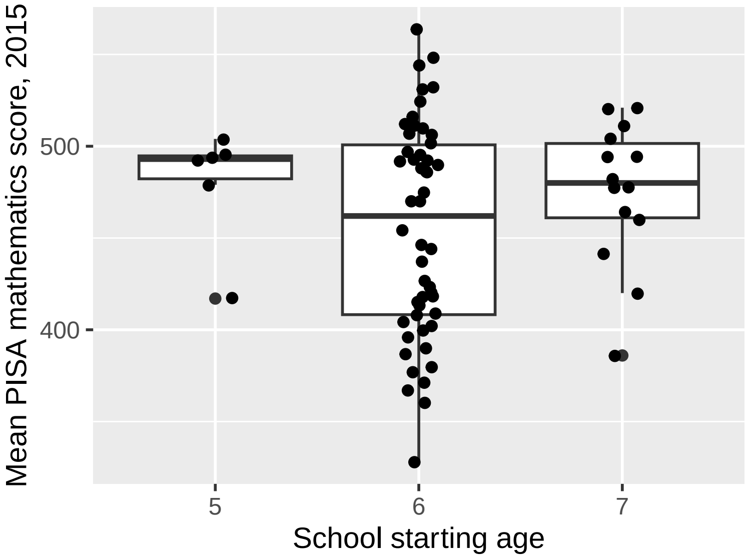
```
ggplot(data = maths, aes(x = as.factor(start.age), y = score)) +
  geom_boxplot() +
  labs(x = "School starting age",
       y =  "Mean PISA mathematics score, 2015")
```



8.4.1 Adding the observations to a box plot

It may be helpful to show the individual observations as well, for example, if the distribution of points within a group is bimodal; this wouldn't be apparent from a box plot. It can help to 'jitter' the points on the group axis so that each point can be seen clearly. This can be done as follows:

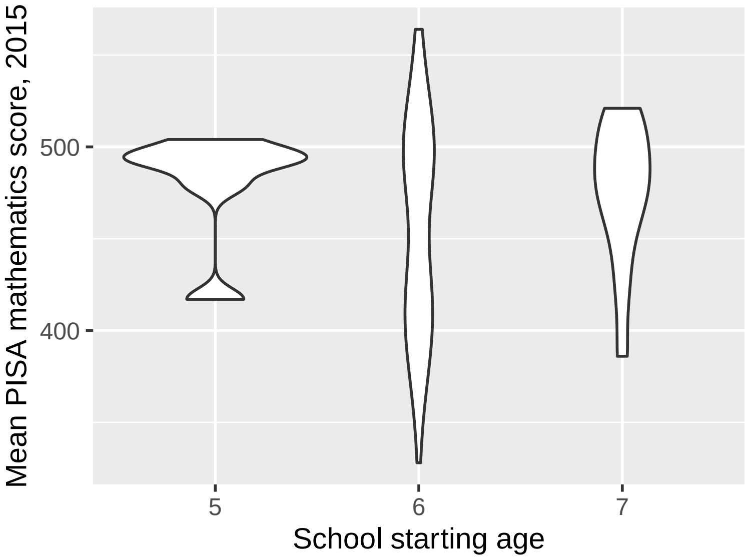
```
ggplot(data = maths, aes(x = as.factor(start.age), y = score)) +
  geom_boxplot() +
  labs(x = "School starting age",
       y =  "Mean PISA mathematics score, 2015") +
  geom_jitter(width = 0.1)
```



8.4.2 Violin plots

An alternative to a box plot is a "violin plot", which plots (mirrored) density estimates for each group.

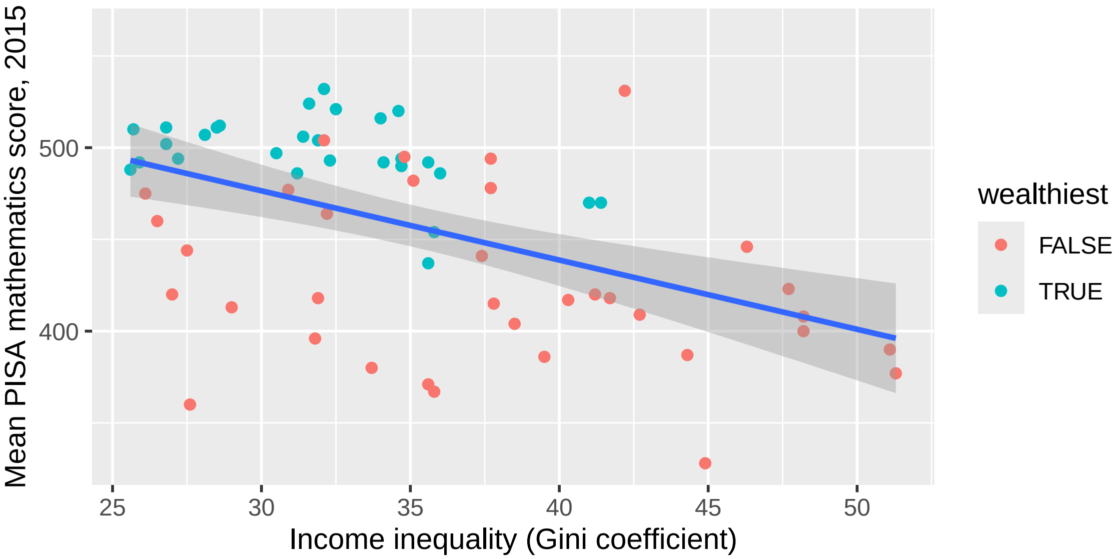
```
ggplot(data = maths, aes(x = as.factor(start.age), y = score)) +
  geom_violin() +
  labs(x = "School starting age",
       y =  "Mean PISA mathematics score, 2015")
```



8.5 ‘Global’ aesthetics

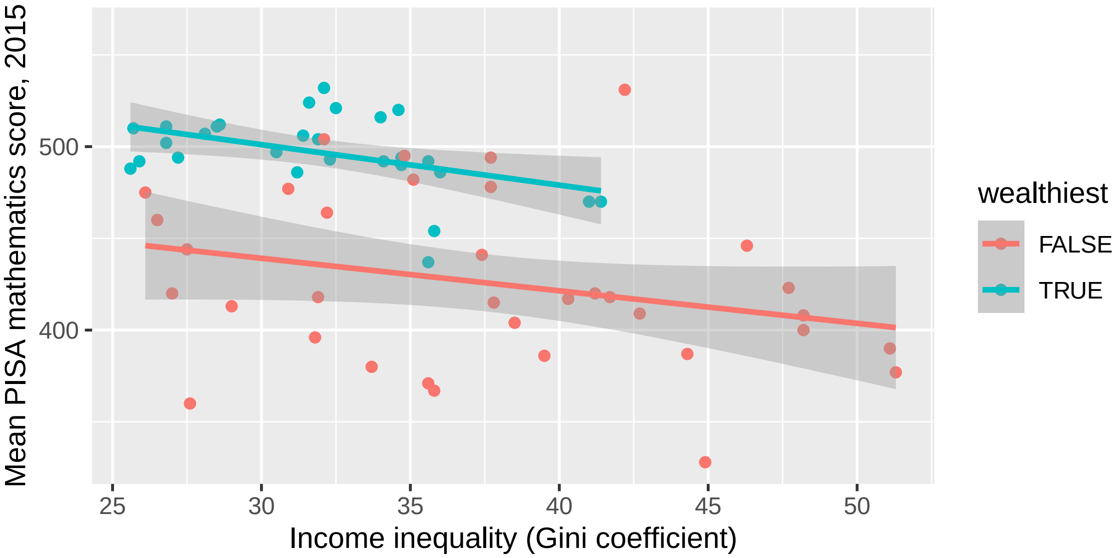
Aesthetics can be defined in different places, and this affects the appearance of the plot. Here’s an example scatter plot, where we map colour to `wealthiest` within the `geom_point()` command:

```
ggplot(data = maths,
       aes(x = gini, y = score)) +
  geom_point(aes(colour = wealthiest)) +
  labs(x = "Income inequality (Gini coefficient)",
       y = "Mean PISA mathematics score, 2015") +
  geom_smooth(method = "lm")
```



Note that the trend specified in `geom_smooth()` refers to a single trend for all the data. If we instead map `wealthiest` to colour in the `ggplot()` command, this is now applied 'globally' in all the subsequent `geom` commands:

```
ggplot(data = maths,
       aes(x = gini, y = score, colour = wealthiest)) +
  geom_point() +
  labs(x = "Income inequality (Gini coefficient)",
       y = "Mean PISA mathematics score, 2015") +
  geom_smooth(method = "lm")
```

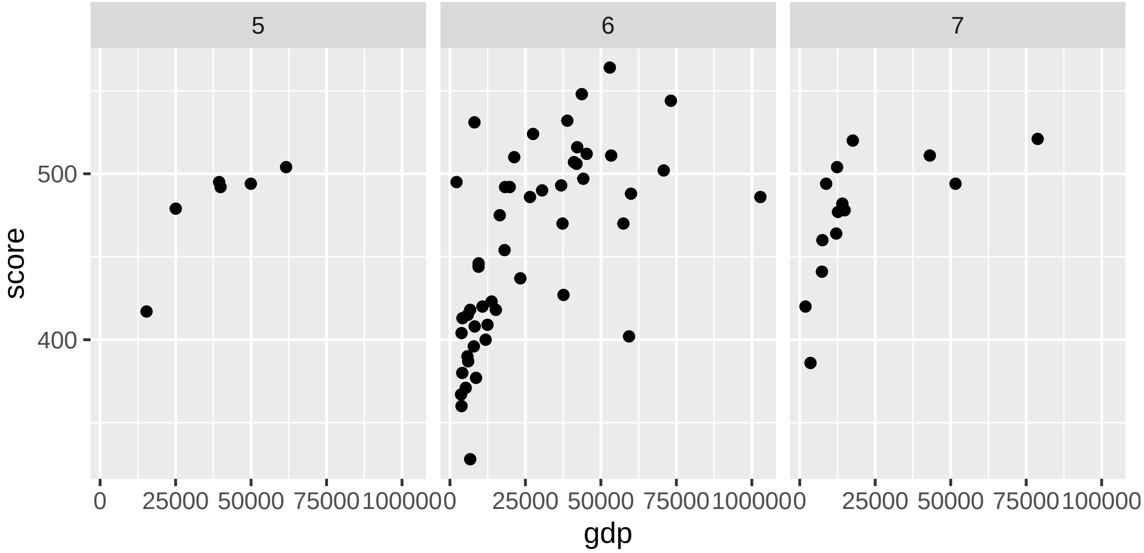


8.6 Facets

In some cases, we may wish to create a figure with multiple plots side-by-side, or arranged in a grid. If the plots are of the same type, with the same axes, we can use *facets*, which will ensure that the scales on each axes are the same, making comparison between the plots easier.

For example, suppose we want a separate scatter plot of `score` against `gdp`, for each value of `start.age`. We can do this as follows:

```
ggplot(maths, aes(x = gdp, y = score)) +
  geom_point() +
  facet_grid(cols = vars(start.age))
```



See `?facet_grid` and also `?facet_wrap` for more details (look at the examples in particular, for the different ways these plots can be laid out).

8.7 Exercises

Exercise 8.1. For this exercise, you will need the `Brexit` data frame (from the file `Brexit.csv`), described in Exercise 6.1.

1. Produce a scatter plot of the percentage voting to remain in the EU (within each local authority), against the percentage of adults with Level 4 qualifications.
    - Specify suitably descriptive axes labels;
    - add a linear trend to your plot;
    - colour each point by its `Region` variable.
2. Produce a histogram showing the distribution of the percentage voting to remain in the EU across the different local authorities.
    - Specify a suitably descriptive axis label;
    - annotate your plot to indicate the vote in Sheffield.
3. Produce a box plot for the percentage voting to remain in the EU within each region.
    - Experiment with ‘horizontal’ and ‘vertical’ box plots.

8.8 Further reading

- There is an RStudio Cheatsheet on `ggplot2`: go to **Help > Cheatsheets > Data Visualization with ggplot2**.

- The R Graphics Cookbook (2nd edition) is a good reference for making plots with `ggplot2`.
- There is an online tutorial available here: RStudio Primers (Visualize Data)
- See also R for Data Science (Chapter 3)

Chapter 9

Presentation of plots

It is important to think carefully about how you present your plots to others. With minimal plotting commands you can obtain a plot quickly and easily, *but it is unlikely it will be suitable for including in a report.*

For this chapter, you will need to install the `gridExtra` package, and a package called `MAS6005` that is available on GitHub only. Install these with the following commands [you only need to install these packages once].

```
install.packages("gridExtra")
install.packages("devtools")
devtools::install_github("OakleyJ/MAS6005")
```

9.1 The basics

Using the `mtcars` data frame in R, we'll make a scatter plot of miles per gallon against weight. For more information about the data and these variables, use `?mtcars`. Suppose we use the following code, and the figure below is used in a written report:

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```

This standard of presentation would be unacceptable in any presentation or report! To improve the figure, we should:

1. include proper axes labels: never simply use the R variable names;
2. specify the units;
3. give sufficiently detailed captions so that the figure can be understood on its own, **and** include a conclusion: what do we learn from the figure?

The first two points are obvious, but the third is perhaps less so, so we'll discuss this a little more.

9.2 The caption test

You may be unsure as to whether you should include a particular plot or not. You may be tempted to 'err on the safe side' by including the plot, but if the plot doesn't tell the reader anything useful

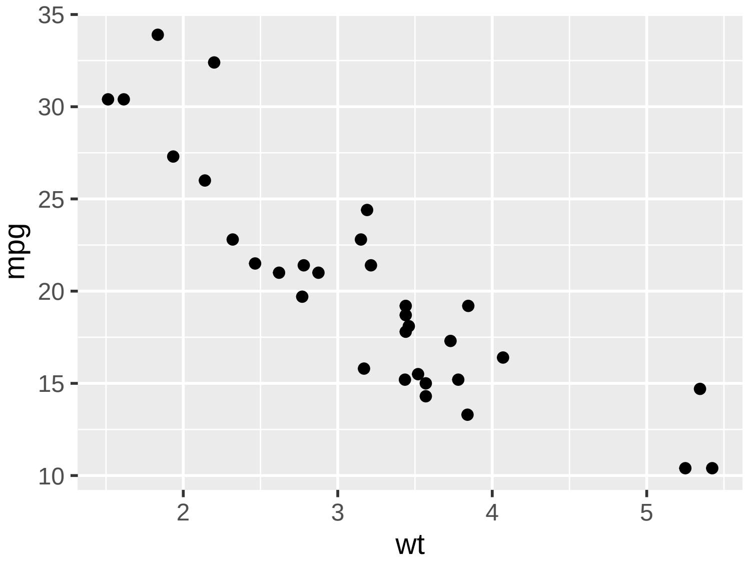


Figure 9.1: Fuel economy and weight.

this will just lead to a bloated report. A simple test to apply is as follows.

> State, in your caption, what **conclusion** the reader should draw from looking at the plot. If you can't think of anything to say, this probably means that there's nothing useful to be learned from your plot: so leave it out of your report!

This doesn't mean, for example, that a plot must show an interesting relationship between two variables; a plot may suggest that two variables are unrelated; that can still be informative to a reader. There will also be exceptions: it may be helpful to include a plot simply to show what data are available, but it should be obvious when you need to apply the caption test.

It's also good practice to state the data source in the caption. So in summary, the caption should include:

1. a title for the plot;
2. a conclusion that can be drawn from the plot;
3. the source of the data, if appropriate.

Putting this all together, an improvement would be

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(x = "Weight (lb/1000)", y = "Miles / (US) gallon")
```

9.3 Caption or title?

In a written report, plots will usually be labelled by a figure number and then the plot information would all go in the caption. It is not recommended to have an *additional* title at the top: you would then have plot text in two different places.

However, if your plot is being presented in a web page, or in some talk slides, then you may not have a figure label and caption. Here, you can use additional `ggplot2` commands to produce a title,

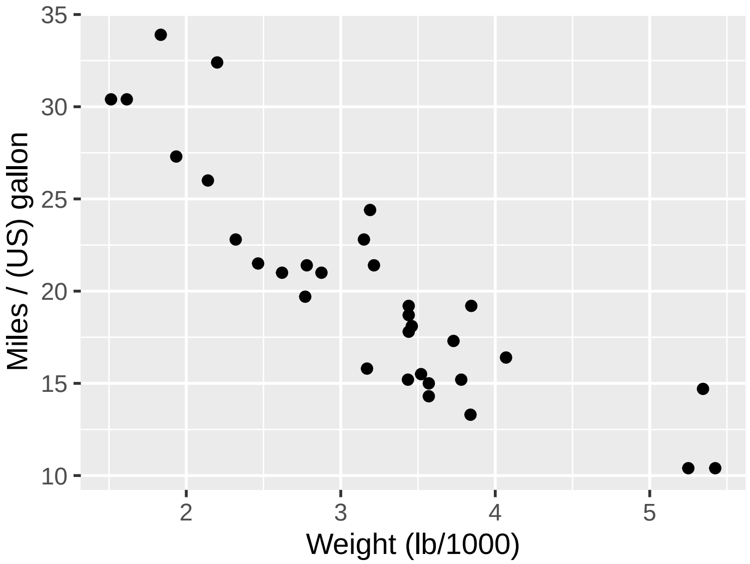
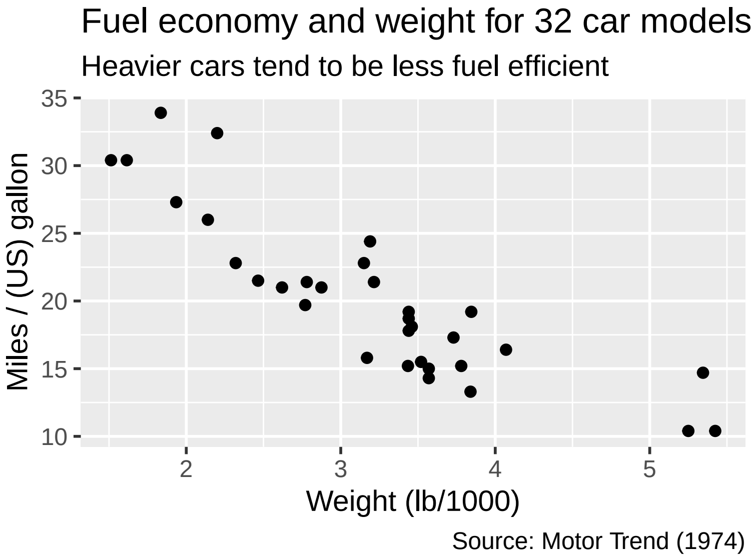


Figure 9.2: Fuel economy and weight for 32 car models. Heavier cars tend to be less fuel efficient. Source: Motor Trend (1974)

conclusion and data reference within the plot itself. For example (with `subtitle` used for the plot conclusion, and `caption` used for the data source),

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  labs(x = "Weight (lb/1000)", y = "Miles / (US) gallon",
       title = "Fuel economy and weight for 32 car models",
       subtitle = "Heavier cars tend to be less fuel efficient",
       caption = "Source: Motor Trend (1974)")
```



9.4 Customising the appearance of a plot

You can change just about any aspect of the appearance of a plot. If you have a legend in your plot, it's likely you'll need to modify it as the default will use data frame column names. You may also wish to change the grey background used by default in `ggplot2`. Font sizes will need changing if they are too small in your final report / presentation.

The R Graphics Cookbook is an excellent reference here. (The format used throughout is to state a "problem": something you want to do with your plot, and then provide the code solution and discussion). Some chapters in particular that may prove helpful to you are:

- Chapter 9 Controlling the Overall Appearance of Graphs,
- Chapter 10 Legends,
- Chapter 12 Using Colors in Plots.

9.5 Refining a plot: an example

We'll now give an example of creating a plot, and then thinking about how we might improve it (assuming we've already got 'the basics' right, in that the axes titles and caption are satisfactory).

9.5.1 The data to plot

We consider the `mvscores` data set from the `MAS6005` package. The aim here is to compare test match batting scores for one player, Michael Vaughan, between the matches he played as captain, and the matches where he was not captain. The hypothesis is that players tend to perform less well if they have the added burden of captaincy.

The data set also records whether each score was in the first or second innings; batting can be more difficult in a second innings, due to wear of the pitch.

To summarise for those with no interest in / knowledge of cricket:

- we want to illustrate how/if the values in `runs` differ depending on the `captain` variable (a 2-level factor: `yes` or `no`)
- `innings` (a 2-level factor: `first` or `second`) is a 'blocking variable': we are more interested in comparing captain/not captain scores within the same innings type than between different innings types.

9.5.2 A first attempt: four histograms

We'll first try producing four histograms of scores: one for each combination of `captain` and `innings`. Note that:

- we use the `gridExtra` package (Auguie, 2017) to arrange the plots in a 2x2 grid;
- plots produced by `ggplot2` can be assigned to variables, and used later in other functions.

```
library(MAS6005)
```

```
p1 <- mvscores %>%
  filter(innings == "first", captain == "yes") %>%
  ggplot(aes(x = runs))+
  labs(x = "First innings runs, captain") +
```

```r
  geom_histogram()

p2 <- mvscores %>%
  filter(innings == "first", captain == "no") %>%
  ggplot(aes(x = runs))+
    labs(x = "First innings runs, not captain") +
  geom_histogram()

p3 <- mvscores %>%
  filter(innings == "second", captain == "yes") %>%
  ggplot(aes(x = runs))+
    labs(x = "Second innings runs, captain") +
  geom_histogram()

p4 <- mvscores %>%
  filter(innings == "second", captain == "no") %>%
  ggplot(aes(x = runs))+
    labs(x = "Second innings runs, not captain") +
  geom_histogram()

gridExtra::grid.arrange(p1, p2, p3, p4, nrow = 2)
```

A key issue with this plot is that the x-axis scales are different for each histogram, which makes comparing the histograms harder. We could set the scale manually (see `?ggplot2::xlim`) but facets might work well here.

The y-axis scales are also different. This issue is slightly more complicated in that the numbers of observations used for each histogram are different, so it's really the shapes of the histograms that we want to compare. One option is to scale each histogram to have total area 1 (so it's like a density plot).

It's also worth thinking about the arrangement of the plots within the grid. Using rows rather than columns to represent the main factor of interest (`captain`), would mean that the main comparisons involve looking at histograms aligned vertically, not horizontally: any 'shift' in distribution is easier to see.

An alternative plot, which addresses these issues (and I prefer), is as follows:

```r
ggplot(mvscores, aes(x = runs, y = ..density..))+
  labs(x = "Runs scored") +
  geom_histogram() +
  facet_grid(rows = vars(captain),
             col = vars(innings),
             labeller = label_both)
```

9.5.3 Using a box plot

Another option is to present the data using a box plot. Although we lose some information compared with the histogram, comparing summaries of the distributions of scores is easier.

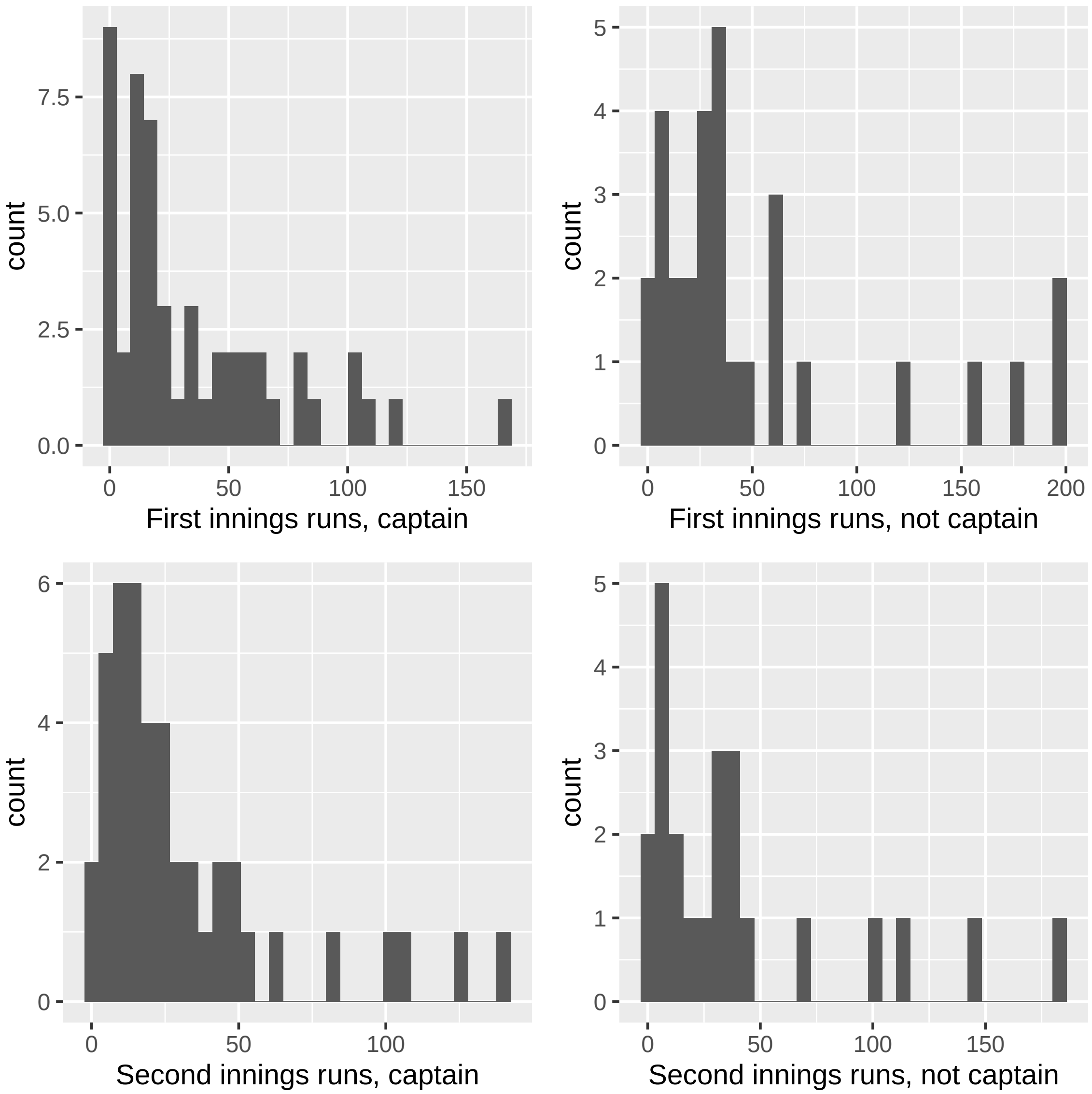


Figure 9.3: Test Match runs scored by the England Cricketer Michael Vaughan, in each innings, and whether or not he played as captain. His scores tended to be higher when he did not play as captain. Source: ESPNcricinfo.

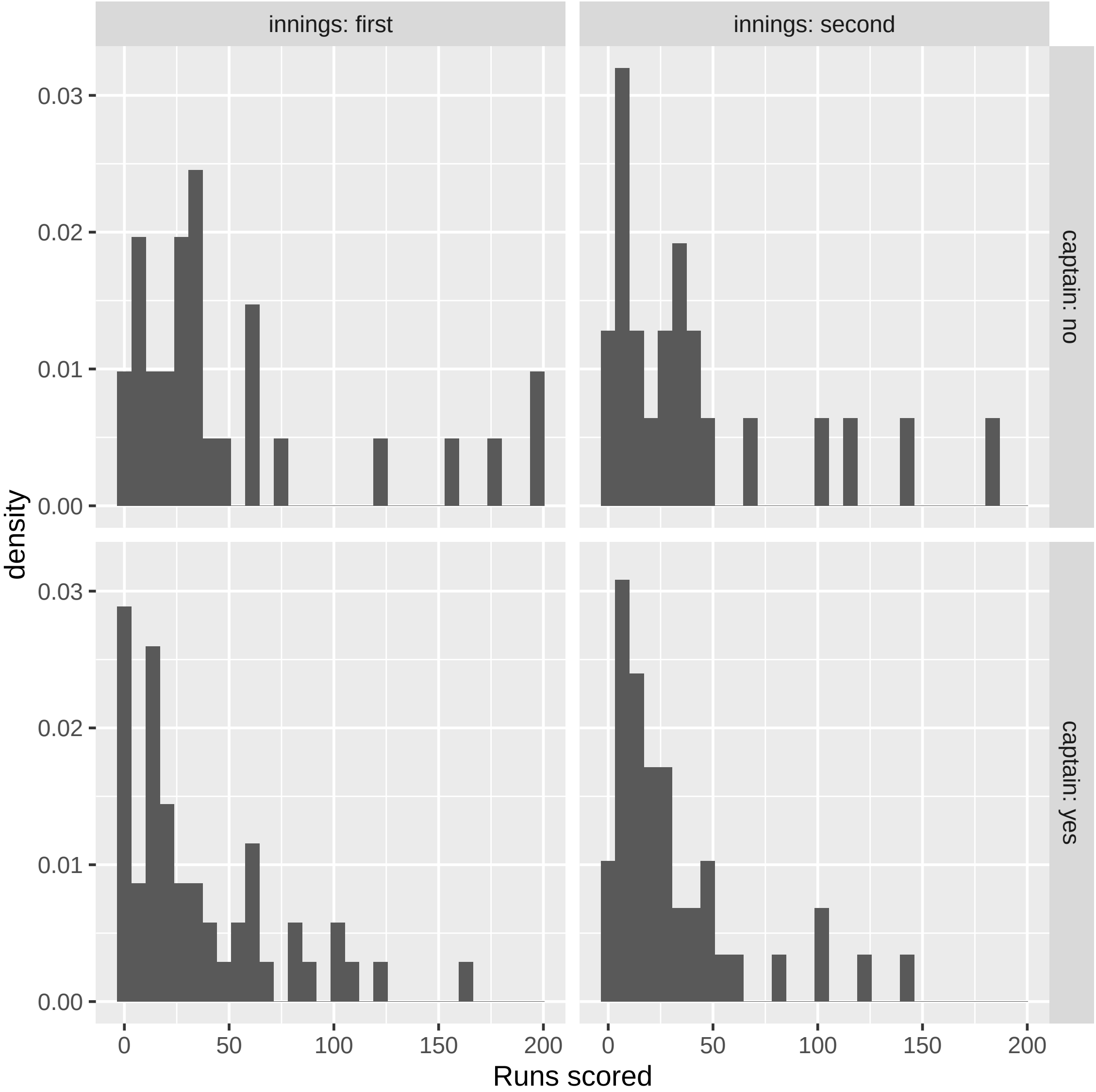


Figure 9.4: Test Match runs scored by the England Cricketer Michael Vaughan, in each innings, and whether or not he played as captain. His scores tended to be higher when he did not play as captain. Source: ESPNcricinfo.

```
ggplot(mvscores, aes(x = innings, y = runs))+
  geom_boxplot(aes(color = captain) )
```

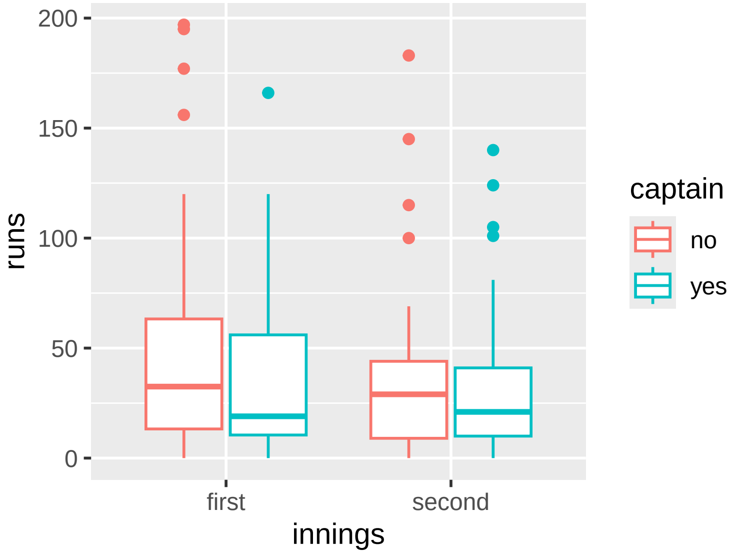


Figure 9.5: Test Match runs scored by the England Cricketer Michael Vaughan, in each innings, and whether or not he played as captain. His scores tended to be higher when he did not play as captain. Source: ESPNcricinfo.

I prefer this to the histogram plot. Note that it's more helpful to map `captain` to colour and `innings` to the x-axis than vice-versa, as this makes it easier to compare the effect of 'treatment' (`captain`) within each 'block' (`innings`).

9.6 Exercises

Exercise 9.1. Below are three plots. For each plot:

- run the code in R to reproduce the plot;
- think about how each plot might be improved;
- modify the R code to achieve a better plot.

Hints are given for each plot, but try not to read them until you have had your own ideas!

1. This plot uses the built in data set `airquality`. Type `?airquality` for more details.

```
ggplot(airquality, aes(x = Temp, y = Ozone)) +
  geom_point()
```

This plot fails on all levels regarding The basics and The caption test! Use the help file so you can specify more informative labels.

2. This following plot uses the `medals` data set from the `MAS6005` package, and shows number of gold medals against population size.

```
ggplot(MAS6005::medals, aes(x = population, y = gold)) +
  geom_point() +
```

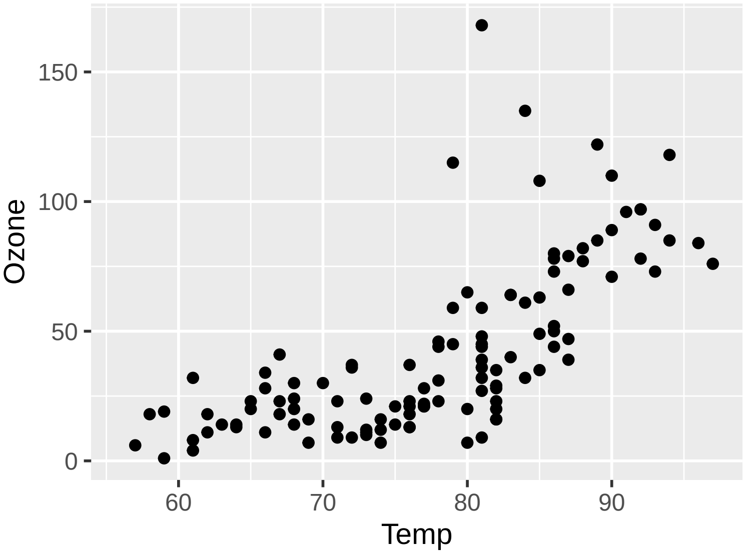


Figure 9.6: Scatter plot of ozone versus temperature

```
  labs(x = "population size", y = "gold medals won")
```

- The caption refers to India and China. Although the reader might guess which points are these two countries, they shouldn't have to! Annotations would help.
- The bunching of most of the points in the bottom left corner doesn't look very nice. A log-scale x-axis is worth trying.
- The scientific notation used for the x-axis scale is unfriendly for the general reader. It might help to express population size in units of millions.

3. The following plot uses the `inequality` data set from the `MAS6005` package, and shows income inequality for different countries.

```
ggplot(MAS6005::inequality, aes(y = country, x = gini)) +
  geom_col() +
  labs(x = "Gini coefficient")
```

- Visualising the rank order is difficult here, as the bars are arranged in alphabetical order of country. Ordering them by Gini coefficient would help. (See this example in the R Graphics Cookbook.)
- The y-axis label "country" is unnecessary here, and can be removed.
- As the caption refers to the UK, we could try to make the UK observation more distinctive in the plot. Here, you could try using the `fill` argument in `geom_cols()`, specifying it to be a vector of 36 colours: 35 the same, and one different for the UK.

9.7 Data sources

- The air quality data was obtained from the New York State Department of Conservation (ozone data) and the National Weather Service (temperature data), and provided in the R `datasets` package.

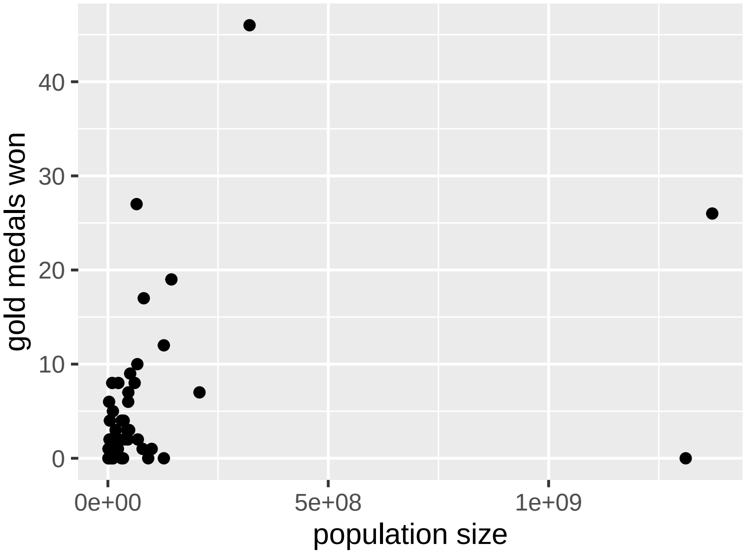


Figure 9.7: Number of gold medals won against population size for the Rio 2016 Summer Olympics. Although India and China have similar population sizes, China was much more successful. Source (population data): World Bank.

- The inequality data was obtained from OECD (2016), Income inequality (indicator). doi: 10.1787/459aa7f1-en [Accessed on 17 August 2016].
- The population data was obtained from The World Bank. Accessed 6th October 2015.
- The medal table data was obtained from https://www.rio2016.com/en/medal-count-country. [Accessed on 6th October 2016, but this link is no longer active.]

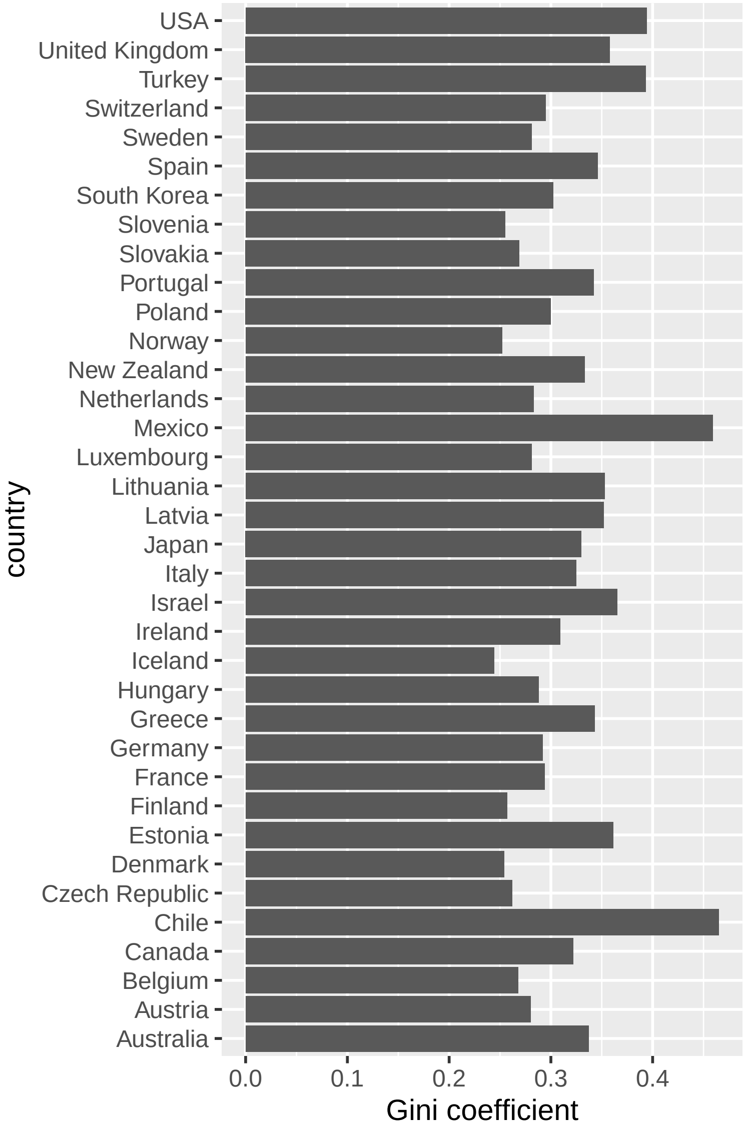


Figure 9.8: Income inequality as measured by the Gini coefficient for 36 OECD countries, reported in 2016. The UK was ranked 7th worst. Source: OECD.

Chapter 10

Maps with `leaflet`

For spatial data corresponding to geographical locations, it can be helpful to visualise the data on a map. There are various mapping tools available in R. Here, we will discuss the R package `leaflet` (Cheng et al., 2019), which is an R implementation of the JavaScript library of the same name.

Maps produced using `leaflet` are best suited for web pages, as you can zoom and scroll a map as you would do with, say, Google maps. These maps can also be used inside shiny apps.

You will need to be online to use `leaflet`.

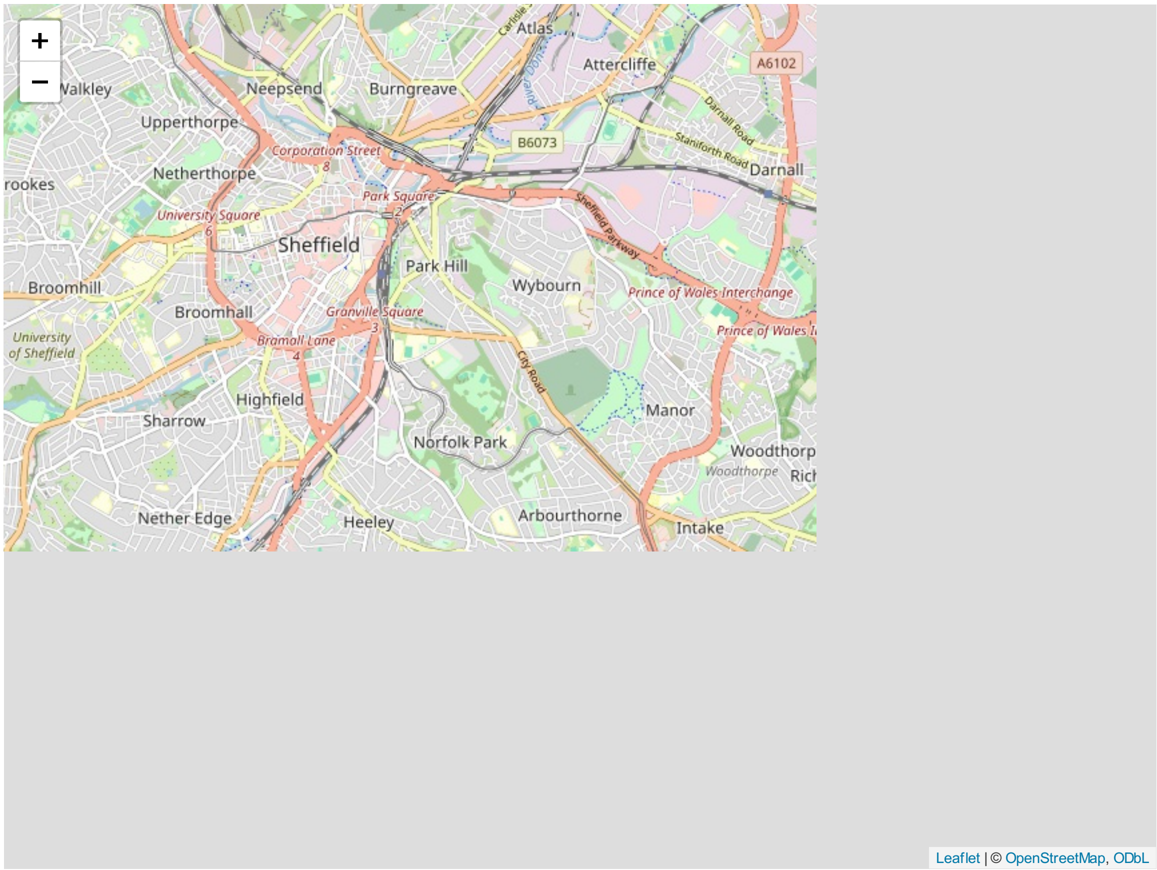
10.1 A basic map

We'll first load the `leaflet` package.

```r
library(leaflet)
```

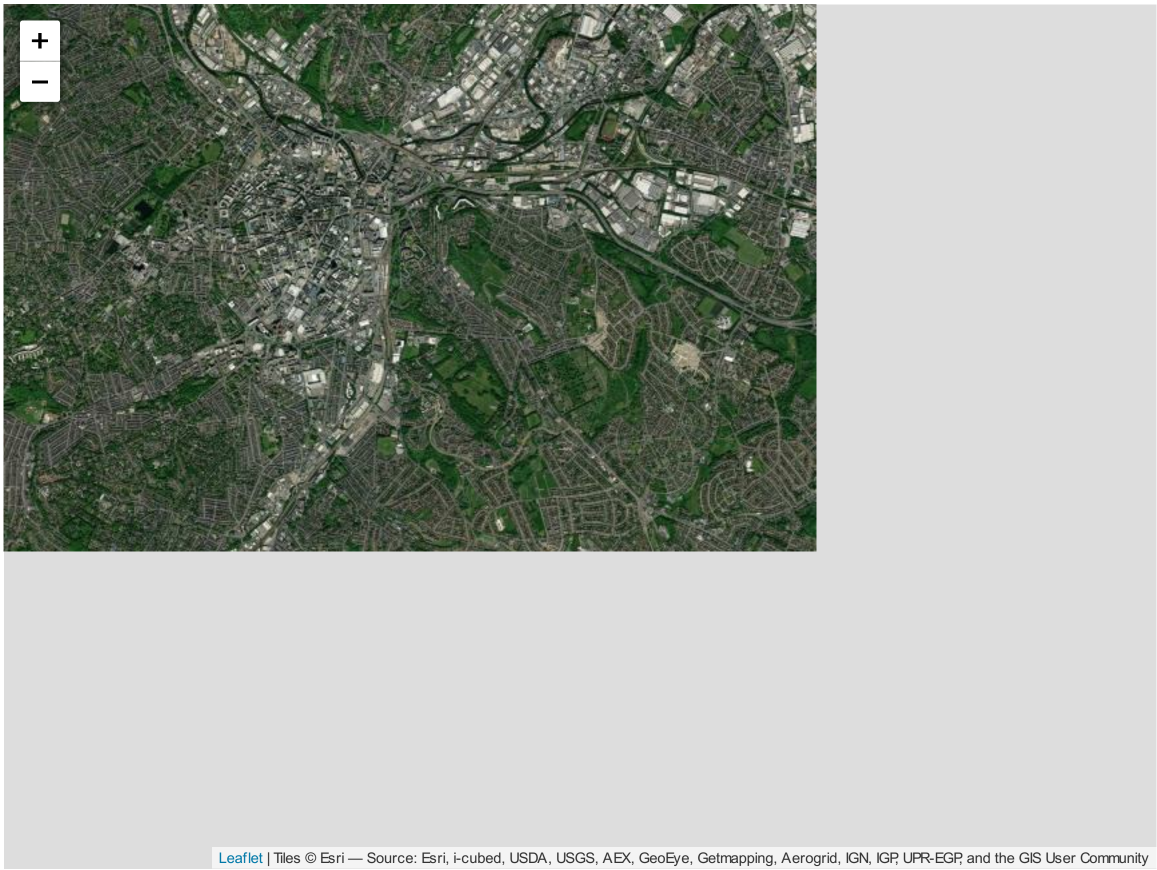
To get a basic map, we will normally need to specify the longitude and latitude of the centre of our map, as well the initial zoom level (which may take some trial and error). For example, to obtain a map of Sheffield:

```r
leaflet() %>%
  addTiles() %>%
  setView(lng = -1.473798,
          lat = 53.38252,
          zoom = 13)
```



The `addTiles()` specifies what is drawn on each map 'tile', with the default being streets and points of interest provided by OpenStreetMap. For example, to change to an aerial photo (as in Google Earth), we can instead use the `addProviderTiles()` command and specify a different 'tile provider':

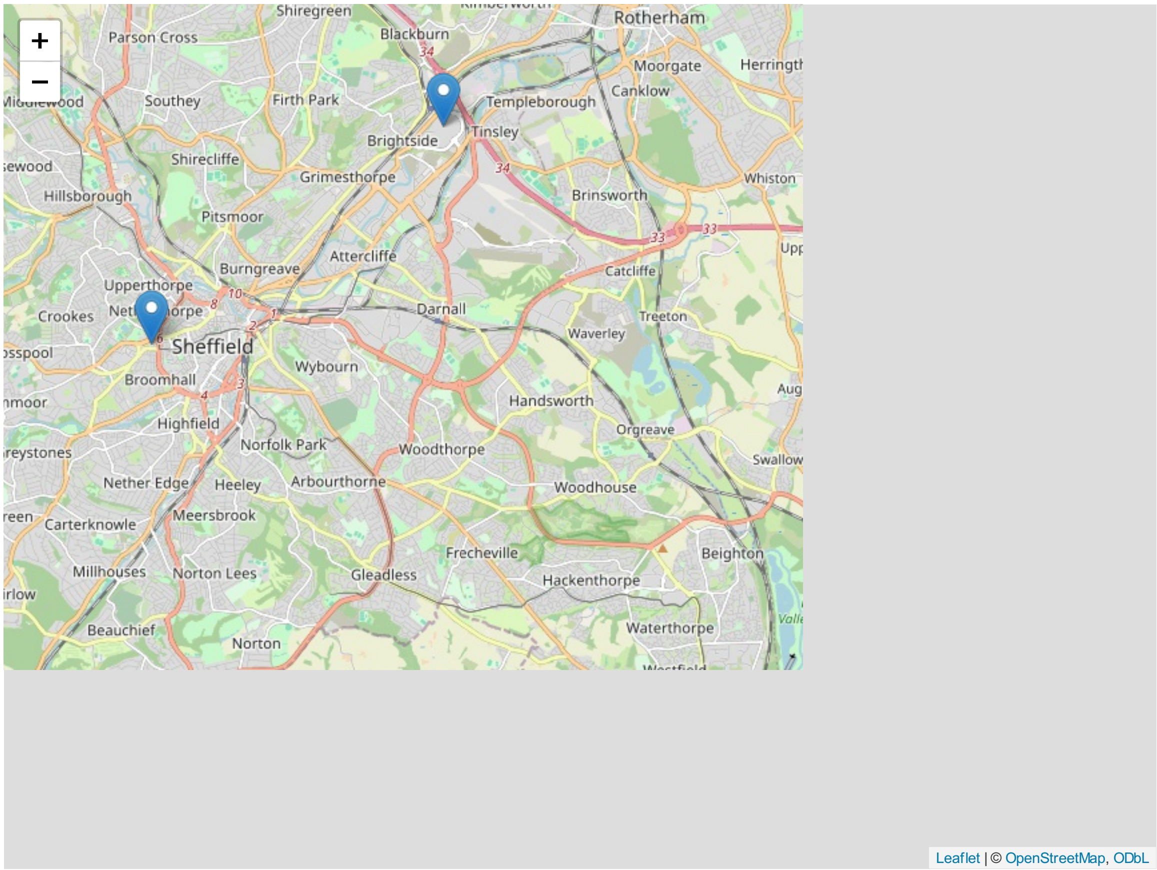
```r
leaflet() %>%
  setView(lng = -1.473798, lat = 53.38252, zoom = 13) %>%
  addProviderTiles(providers$Esri.WorldImagery)
```


+
−
Leaflet | Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

10.2 Indicating points on a map

If we have vectors of coordinates (latitude and longitude), we can add points with the `addMarkers()` function (there are variants such as `addCircles()` and `addPopups()`). We can include the argument `popup` so that some text is displayed if the mouse is clicked on the marker. Here's an example where I've marked the Hicks Building and Meadowhall:

```r
latitude <- c(53.3809, 53.4143)
longitude <- c(-1.4862, -1.4109)
placeNames <- c("Hicks Building", "Meadowhall")
leaflet() %>%
  setView(lng = -1.45, lat = 53.40, zoom =12) %>%
  addTiles() %>%
  addMarkers(lat = latitude,
             lng = longitude,
             popup = placeNames)
```



10.3 Exercise

Exercise 10.1. Use leaflet to make a map that displays a region and two points of interest to you. For example, where you live and your nearest railway station.

10.4 Further reading

- The `leaflet` R package is well documented: a manual is available here.
- Although not necessary for using the R package, if you want to understand more about leaflet itself, documentation and tutorials are available at The JavaScript leaflet homepage.

Part III

Statistical analysis

Chapter 11

Computing using loops

Sometimes we may be working with data where we require to repeat a block of code multiple times. For example, reading in data sets spread across multiple files where the structure of the data within each file is the same, or, performing a set of calculations for different groupings of the data. It can be tempting to copy and paste a block of R code, once for each occurrence, editing each block as necessary. Try to avoid this if you can! Your code may get messy/hard to read if you have lots of occurrences, and it may lead to bugs if you don't edit each block correctly.

11.1 Repeating a process with a `for` loop

There are different ways we might get R to run the same block of code multiple times (with small changes each time). It may be a good idea to create your own function, but we'll consider a simpler solution here, which is to put code inside a `for` loop. (There are more efficient methods, but `for` loops are easy to write and read, and will be sufficient for this module.)

A `for` loop has the basic syntax

```
for(i in 1:n){

}
```

- Everything inside the curly brackets will be carried out `n` times;
- any instance of `i` will be replaced by `1`, then `2`, `3`,…,`n`. If, for example, there is an `x[i]` inside the curly brackets, this code will be run first using `x[1]`, then using `x[2]` and so on.

11.2 Example: cleaning two text files

As an example, we'll use the fictitious student data, but now suppose there are two data files: `stat101.txt` and `stat102.txt`. We suppose each data set needs cleaning, and then we'd like to combine the two data frames.

11.2.1 The core code block

We have a 'core' block of code that we want to use multiple times, making small changes each time. The code to get a single data file into a data frame was as follows. (We'll add an extra `mutate()` command to store the module code.)

```
examTextRaw <- read_lines("data/stat101.txt")

examTextClean <- examTextRaw %>%
  str_remove_all(pattern = "\\*") %>%
  str_replace_all(pattern = "--",
                  replacement = "NA") %>%
  str_trim()

header <- str_which(examTextClean, pattern = "student")
endLine <- str_which(examTextClean, pattern = "denotes")

read_table(examTextClean[header:(endLine - 1)]) %>%
  mutate(module = "stat101")
```

We want to run this block twice, once for the file `stat101.txt` and once for `stat102.txt`.

11.2.2 Identify the variables we need to specify

The code block above needs to run twice, once using the file name `stat101.txt`, and once using the name `stat102.txt`. We will have to specify these in advance. We can actually just specify module codes:

```
modules <- c("stat101", "stat102")
```

and then construct the file paths using `paste0()`:

```
paste0("data/", modules, ".txt")
```

```
## [1] "data/stat101.txt" "data/stat102.txt"
```

11.2.3 Create an empty list to store the results

Lists are useful here, as an element of a list can be any type of object. We make an empty one as follows:

```
moduleResults <- vector(mode = "list", length = length(modules))
moduleResults
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
```

(We could have just specified `length = 2`, but try to avoid specifying numerical values like this if they may change at some point, e.g. if another module was to be added.)

11.2.4 Use a for loop to run the code block multiple times

The code to be repeated goes inside a for loop, with one element of the list `moduleResults` filled each time. (We'll repeat the commands to set up the variables at the start.)

```
modules <- c("stat101", "stat102")
filePaths <- paste0("data/", modules, ".txt")
moduleResults <- vector(mode = "list", length = length(modules))

for(i in 1:length(modules)){
  examTextRaw <- read_lines(filePaths[i])

  examTextClean <- examTextRaw %>%
    str_remove_all(pattern = "\\*") %>%
    str_replace_all(pattern = "--",
                    replacement = "NA") %>%
    str_trim()

  header <- str_which(examTextClean, pattern = "student")
  endLine <- str_which(examTextClean, pattern = "denotes")

  moduleResults[[i]] <-
    read_table(examTextClean[header:(endLine - 1)]) %>%
    mutate(module = modules[i])
}
rm(i, examTextRaw, examTextClean, header, endLine)
```

> The remove command `rm()` at the end just tidies up the background R environment. [When a `for` loop is run outside of a function, anything defined in the loop will be retained in the background R environment, evaluated at the values used in the final iteration of the loop.]

11.2.5 Convert the list to a data frame

As each data frame in the list has the same column headings, we can convert the list to a data frame as follows:

```
do.call(rbind.data.frame, moduleResults)
```

```
## # A tibble: 7 x 4
##   student   cwk  exam module
##     <dbl> <dbl> <dbl> <chr>
## 1   12015    55    62 stat101
## 2   12468    78    84 stat101
## 3   11560    55    40 stat101
## 4   12589    62    NA stat101
## 5   12015    61    69 stat102
## 6   12468    81    78 stat102
## 7   11579    51    40 stat102
```

11.3 Example: averaging over groups

Here, we will continue to use the `maths` data example, and use a `for` loop to compute the average (mean) test score across countries for each continent. Let's read in the data:

```
library(tidyverse)
maths <- read_csv("https://oakleyj.github.io/exampledata/maths.csv") %>%
  mutate(wealthiest = gdp > 17000)
```

11.3.1 The core code block

To compute the mean test score for a particular continent, e.g. Europe, we must first filter the data to extract the data of all countries in Europe, and then compute our mean statistic for that data. Hence, our 'core' block of code that we want to use multiple times, making small changes each time, is as follows:

```
Cont_data <- maths %>% filter(continent=="Europe")
mean(Cont_data$score)
```

11.3.2 Identify the variables we need to specify

The code block above needs to run for each unique element of the continent column in the `maths` data frame: Africa, Asia, Europe, North America, Oceanea and South America. We must specify these in advance:

```
Continents <- unique(maths$continent) %>% sort()
Continents
```

```
## [1] "Africa"        "Asia"          "Europe"        "North America"
## [5] "Oceanea"       "South America"
```

11.3.3 Create an empty vector to store the results

Here, we will create a single mean value for each continent. Hence we require a vector to store these numerical values, where position `[i]` of the vector corresponds to the continent defined at `Continents[i]` in our `Continents` vector. We specify an empty vector for this output storage as follows:

```
MeanScore <- vector("numeric", length = length(Continents))
MeanScore
```

```
## [1] 0 0 0 0 0 0
```

11.3.4 Use a `for` loop to run the code block multiple times

The code to be repeated goes inside a for loop, with one element of the vector `MeanScore` filled each time. (We'll repeat the commands to set up the variables at the start.)

```
Continents <- unique(maths$continent) %>% sort()
MeanScore <- vector("numeric", length = length(Continents))

for(i in 1:length(Continents)){
  Cont_data <- maths %>% filter(continent==Continents[i])
  MeanScore[i] <- mean(Cont_data$score)
}
rm(i, Cont_data)
```

11.3.5 Convert the vector to a data frame

For tidiness and clarity we now combine the resulting output vector of means with the continent names into a data frame (tibble) as follows:

```
ContMean <- tibble(continent = Continents, MeanScore = MeanScore)
ContMean
```

```
## # A tibble: 6 x 2
##   continent     MeanScore
##   <chr>             <dbl>
## 1 Africa             364.
## 2 Asia               471.
## 3 Europe             476.
## 4 North America      423.
## 5 Oceanea            494.
## 6 South America      401.
```

> The task in this example is to compute a simple summary and the result could be achieved more elegantly by chaining together the `group_by()` and `summarise()` commands, as explained in the [Computing summaries per group] section. When the computations per group become more complex, the use of a `for` loop becomes a more appropriate method.

11.4 Exercise

Exercise 11.1. The data file `Monthly_Wind_MA_1980-95.csv` contains monthly mean wind speed measurements (m/sec) from a weather station near Manchester Airport over the period 1980 - 1995.

1. Read in the data and make a line plot of the monthly mean wind speed data over time. What do you notice about the data? [The `geom` command for a line plot is `geom_line()`; consider the scale of your x-axis here - you will need to create an appropriate variable for the time scale of the data.]

2. Using a `for` loop, compute the annual average mean wind speed for each year over the period of the data and store the result as a data frame with two columns: 'Year' and 'Annual_Mean'.

3. Plot your calculated annual mean wind speed data as a second line plot. Comparing with your plot in part 1., what can you say about the effect of the averaging on the variability in the data?

Chapter 12

Relationships in data

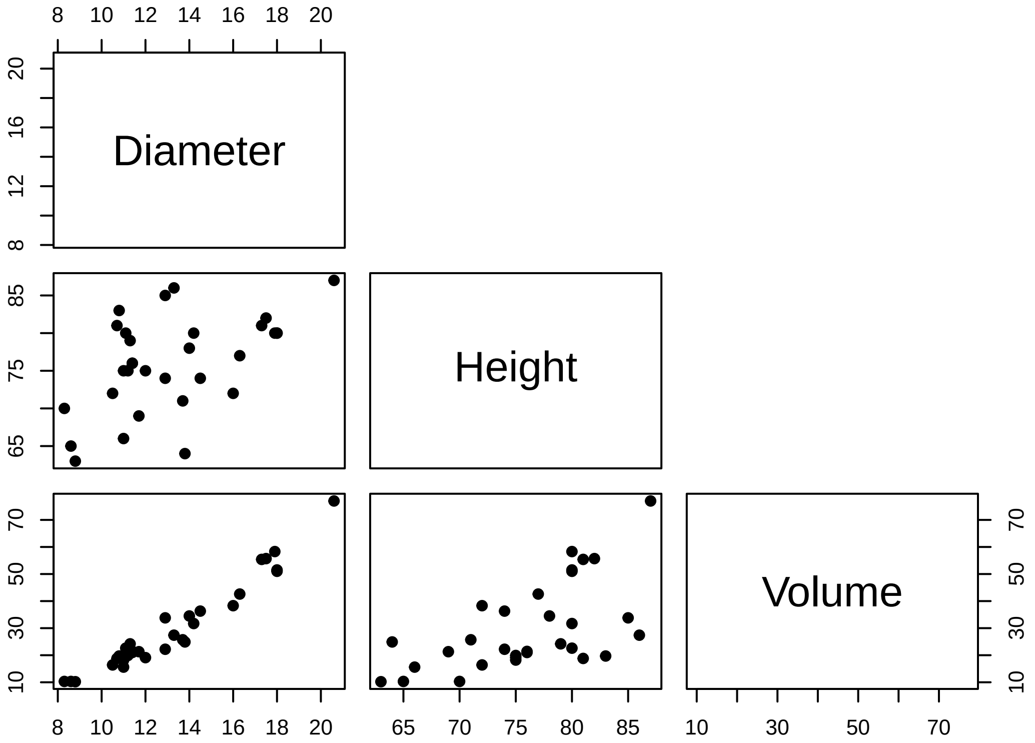
In this section, let's consider a built-in data frame in R called `trees` that contains measurements of the height (in ft), volume (in cubic ft) and diameter (in inches; measured at 4ft 6in above the ground) of the timber in 31 felled black cherry trees. We can load the data into the R environment as follows. (Note: the help documentation for this data set [at `help(trees)`] indicates that the diameter is erroneously labelled 'Girth' in the data frame in R - we will correct this here.)

```
data(trees)
colnames(trees) <- c("Diameter","Height","Volume")
head(trees)
```

```
##   Diameter Height Volume
## 1      8.3     70   10.3
## 2      8.6     65   10.3
## 3      8.8     63   10.2
## 4     10.5     72   16.4
## 5     10.7     81   18.8
## 6     10.8     83   19.7
```

An initial exploration for any relationships between variables in a data set can be made via plotting. Here, we will use pairwise scatter plots to explore for pairwise relationships in the trees data, using the 'base graphics' command `pairs()`:

```
pairs(trees, upper.panel=NULL, pch=19)
```



Clearly, the `Diameter` and `Volume` of the trees has the strongest relationship, and the pairwise relationships between these variables and `Height` are weaker. In all cases, the relationships look to be relatively linear. Let's now look into how we can quantify these relationships more formally.

12.1 Covariance and correlation - recap

Let's first consider the relationship between the diameter and volume of the trees. For the i-th tree, let x_i be its diameter (in inches) and y_i its volume (in cubic ft), for $i = 1, \ldots, 31$. So, we have paired observations $(x_1 = 8.3, y_1 = 10.3)$, $(x_2 = 8.6, y_2 = 10.3)$, $(x_3 = 8.8, y_3 = 10.2)$ and so on.

12.1.1 Covariance

Definition 12.1 (Covariance)**.** For pairs of observations $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ we define their sample covariance to be

$$s_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}),$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.

12.1.1.1 Calculating a covariance in R

In R, we can use the command `cov()` to calculate covariance. For example:

```r
cov(trees$Diameter, trees$Volume)
```

```
## [1] 49.88812
```

and so $s_{xy} = 49.9$ to 1 d.p.

12.1.2 Pearson's correlation coefficient

Covariances aren't very informative on their own, as they will depend on the scale of measurement of the variables. **Correlation coefficients** are scale independent. There are different versions of the correlation coefficient.

Definition 12.2 (Pearson's correlation coefficient)**.** For pairs of observations $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ we define Pearson's correlation coeffcient to be

$$r_{xy} = \frac{s_{xy}}{s_x s_y},$$

with s_{xy} the covariance defined above, and

$$\begin{aligned} s_x &= \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}, \\ s_y &= \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2}. \end{aligned}$$

Pearson's correlation coefficient measures the strength of the **linear** association between the two variables and is bounded between -1 and 1. A positive correlation implies that as one quantity increases, the other is expected to increase, and a negative correlation implies that as one quantity increases, the other is expected to decrease.

12.1.2.1 Calculating Pearson's correlation coefficient in R

To calculate Pearson's correlation coefficient between the variables `Diameter` and `Volume` in the data frame `trees`, we use the command

```r
cor(trees$Diameter, trees$Volume)
```

```
## [1] 0.9671194
```

and so $r_{xy} = 0.97$ to 2 d.p. This value is very large and positive in sign (very close to 1), and suggests a strong positive relationship between these two variables.

12.1.3 Spearman's correlation coefficient

An alternative to Pearson's correlation coefficient is **Spearman's** correlation coefficient.

Definition 12.3 (Spearman's correlation coefficient)**.** For pairs of observations $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ we define Spearman's correlation coefficient to be Pearson's correlation coefficient calculated on the the **ranks** of observations.

By using the ranks of the observations, Spearman's correlation coefficient identifies the strength of any **monotonic** relationship in the data.

12.1.3.1 Example

For illustration, suppose we have the following data

i	1	2	3	4	5	6
x_i	68	2	40	20	85	97
y_i	73	26	37	1	63	68

We first calculate the ranks of the observations (if x_i gets a rank of 1, it means x_i was the smallest out of $x_1, \ldots, x_n$):

i	1	2	3	4	5	6
rank(x_i)	4	1	3	2	5	6
rank(y_i)	6	2	3	1	4	5

We then calculate Pearson's correlation coefficient on the ranks, as if we have six pairs of observations (4, 6), (1, 2), … (6, 5).

12.1.3.2 Calculating Spearman's correlation coefficient in R

In R, we just include an extra argument in the `cor()` command:

```
x <- c(68, 2, 40, 20, 85, 97)
y <- c(73, 26, 37, 1, 63, 68)
cor(x, y, method = 'spearman')
```

```
## [1] 0.7714286
```

(If we don't specify a `method`, the default is to use Pearson's.) To illustrate that this is just Pearson's correlation coefficient calculated on the ranks, we can obtain the rankings in R with the command `rank()`, and then compare the above with

```
rank(x)
```

```
## [1] 4 1 3 2 5 6
```

```
rank(y)
```

```
## [1] 6 2 3 1 4 5
```

```
cor(rank(x), rank(y))
```

```
## [1] 0.7714286
```

12.1.4 Correlations for the `trees` data set

We calculate the Pearson correlations between the variables of interest using the following code:

```
trees %>%
  na.omit() %>%
  cor(method = "pearson") %>%
  round(2)
```

- The second line excludes any rows with missing values (the `cor()` command won't work otherwise);
- the third line will produce a matrix of Pearson correlations, in the form above;
- the fourth line rounds all the numbers to two decimal places.

The code above produces the following output:

```
##          Diameter Height Volume
## Diameter     1.00   0.52   0.97
## Height       0.52   1.00   0.60
## Volume       0.97   0.60   1.00
```

Note that the correlation of any variable with itself is always 1. By looking at these correlations (in absolute value), tentatively, we may conclude the following:

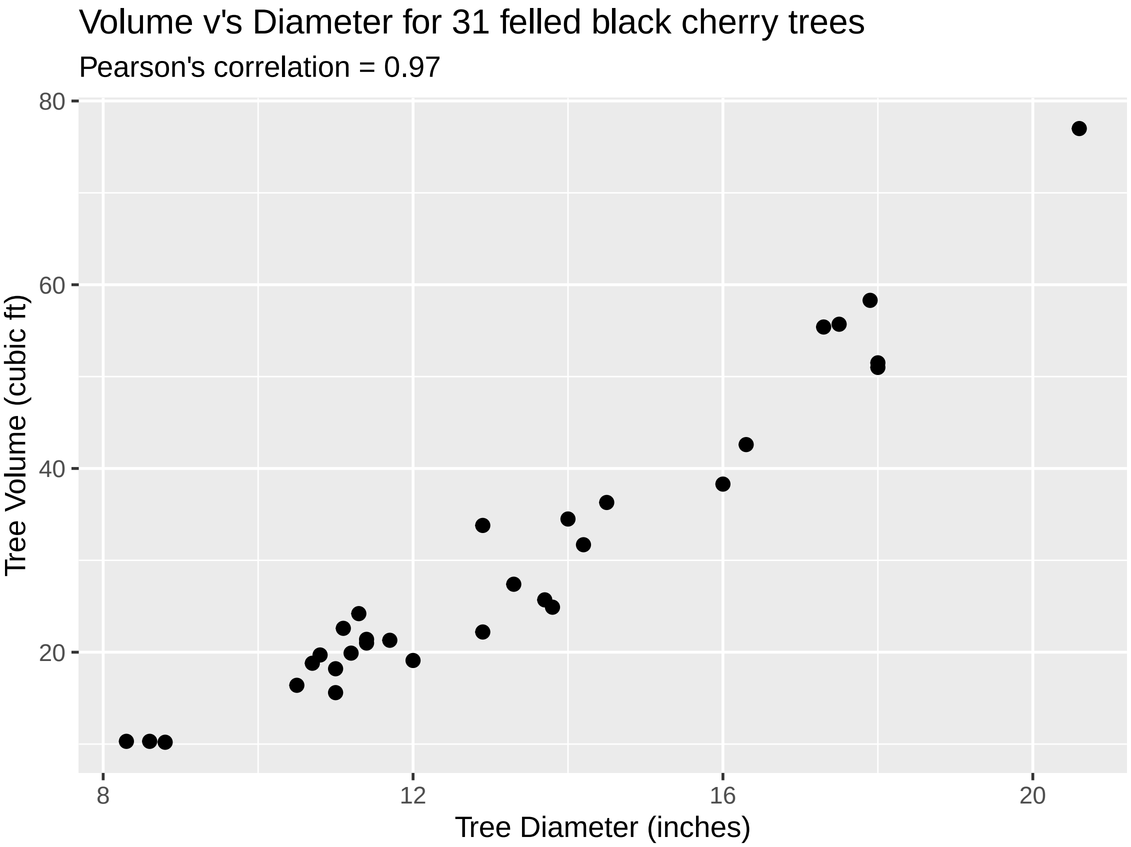
- trees with a larger diameter are likely to have a larger volume (correlation of 0.97);
- taller trees tend to have a larger diameter (correlation of 0.52) and a larger volume (correlation of 0.60).

These correlations match to the inferences we made using the pairwise scatter plots above. (The correlations are all fairly similar if we use Spearman's correlation instead.)

12.2 Simple linear regression

Our scatter plots and correlation values indicate to us that we have a linear relationship between the `Diameter` (x) and `Volume` (y) of the black cherry trees:

```
library(tidyverse)
ggplot(data = trees, aes(x = Diameter, y = Volume)) +
  geom_point(size = 2) +
  labs(x = "Tree Diameter (inches)", y = "Tree Volume (cubic ft)",
       title = "Volume v's Diameter for 31 felled black cherry trees",
       subtitle = paste0("Pearson's correlation = ",round(cor(trees$Diameter,trees$Volume),2))
```



However, this doesn't give us any information about what the straight line of the relationship actually is. **How** is the `Volume` related to the `Diameter`? To explore this, we can 'model' the relationship using a **simple linear regression model** (a linear model with one independent variable) to evaluate the linear relationship in detail.

This linear model is defined as follows:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

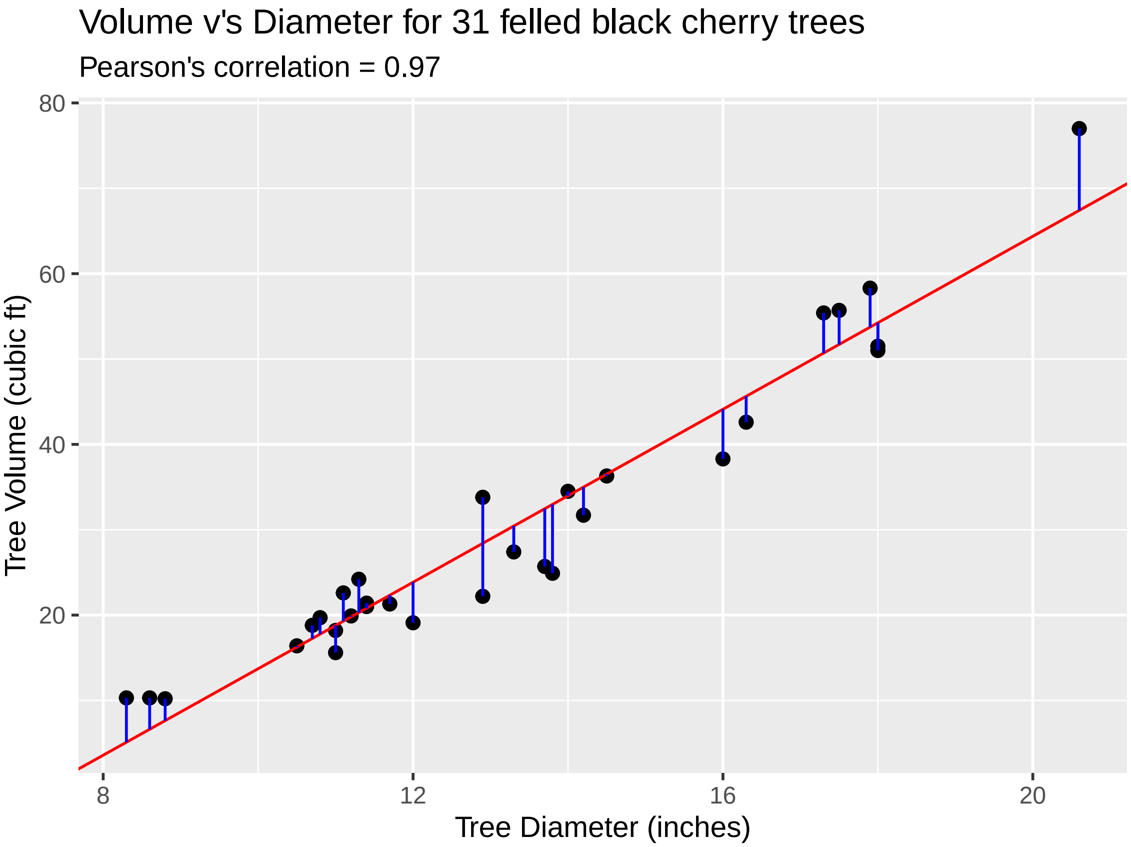
where:

- the x-variable (`Diameter`) is called the **independent** variable with observed values x_i;
- the y-variable (`Volume`) is called the **dependent** variable with observed values y_i;

** **We will use the independent variable to predict the dependent variable.** **

- β_0 and β_1 are the model parameters:
  - β_0 is the **intercept** of the straight line (y, at $x = 0$);
  - β_1 is the **slope** of the straight line (the change in the dependent variable y for every unit change in the independent variable, x);
- ϵ is a **residual error** term (we assume that y is related to x with some random error).

To 'fit' the model to the data we must estimate the parameters β_0 and β_1 from the data to obtain $\hat{\beta}_0$ and $\hat{\beta}_1$. The most common approach is estimation via 'least squares', where the values $\hat{\beta}_0$ and $\hat{\beta}_1$ are selected such that the fitted line minimises the sum of squared residual errors to the data points. The plot below illustrates this method:



The parameter estimates are selected such that for the fitted line they correspond to (shown in red), the sum of the squared residuals (the distances from the data points to the fitted line for the dependent variable y, shown in blue, squared) is minimised.

It turns out that there is an analytical solution for the parameter estimates here:

$$\hat{\beta}_1 = \frac{s_{xy}}{s_x^2} = r_{xy}\frac{s_y}{s_x} \qquad \text{and} \qquad \hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x}$$

where s_x, s_y, s_{xy} and r_{xy} are as defined above in the Pearson's correlation coefficient section. (See MPS223 - Statistical Inference and Modelling, Semester 2 for the full details of this result.)

12.2.1 Fitting the simple linear regression model in R

In practise, we can use R to fit the simple linear regression model and obtain the parameter estimates $\hat{\beta}_0$ and $\hat{\beta}_1$. To fit this model in R, we use the `lm()` command:

```
trees_lm <- lm(Volume ~ Diameter, data=trees)
```

The first argument of this command holds the details of our model specification and is in the format of a 'formula': $y \sim x$, which states that we want to model/predict our dependent variable y (`Volume`) as a linear function of our independent variable x (`Diameter`).

We can view the full details of the fitted model using the summary command:

```r
summary(trees_lm)
```

```
##
## Call:
## lm(formula = Volume ~ Diameter, data = trees)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.065 -3.107  0.152  3.495  9.587
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
## Diameter      5.0659     0.2474   20.48  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.252 on 29 degrees of freedom
## Multiple R-squared:  0.9353, Adjusted R-squared:  0.9331
## F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

Let's briefly examine the output here:

- At the top the output repeats the `Call`, defining the model we have asked it to fit.
- Then, it provides information of the `Residuals`.
- Then, we have the information on the `Coefficients` - our parameter estimates for $\hat{\beta}_0$ ('intercept') and $\hat{\beta}_1$ ('Diameter') [R will use the independent variable's name in the call here]). In this table:
  - the first column holds the parameter estimates;
  - the second column provides an uncertainty estimate for each parameter estimate (these should be small compared to the estimate);
  - then the final 2 columns show the results of t-test's on the importance of the inclusion of each parameter estimate in the model.

- Finally, at the bottom we have more information on the goodness of fit of the model. Particularly, the value of R^2 (labelled `Multiple R-Squared`) indicates the percentage of the variance in the data of the dependent variable that is explained by the model.

> For a simple linear regression model, the value of R^2 is related to the Pearson's correlation co-efficient: $R^2 = r_{xy}^2$.
> [This is **not** true for a more complex linear model that contains more than 1 independent variable. In that case R^2 still exists, but it is no longer the same as the correlation as you have more dimensions, and so are no longer fitting a straight line.]

From the output above, we can state the following:

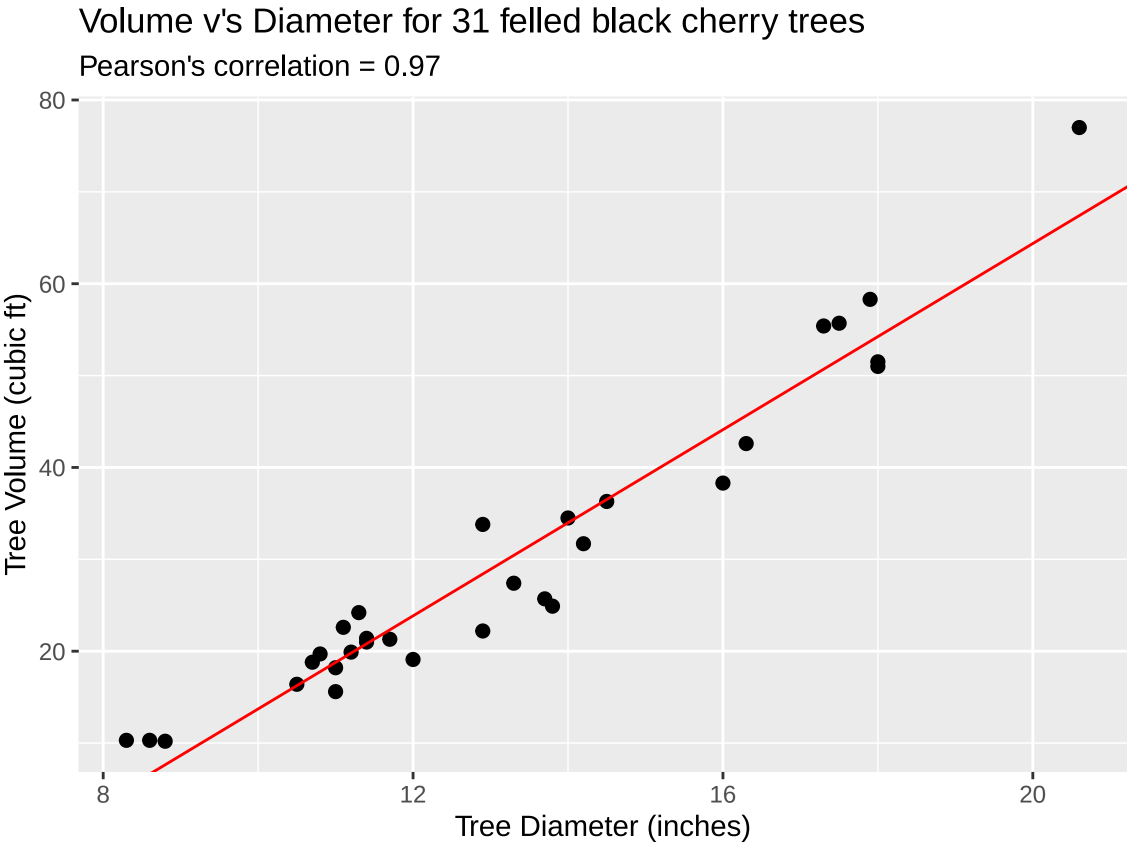
- $\hat{\beta}_0 = -36.9$ with error 3.37.
- $\hat{\beta}_1 = 5.07$ with error 0.25.
- Both $\hat{\beta}_0$ and $\hat{\beta}_1$ are needed in the model.
- The model explains 94% of the variance in the output, Volume.

- The fitted regression line from our linear model is:

$$y_i = -36.9 + 5.07x_i$$

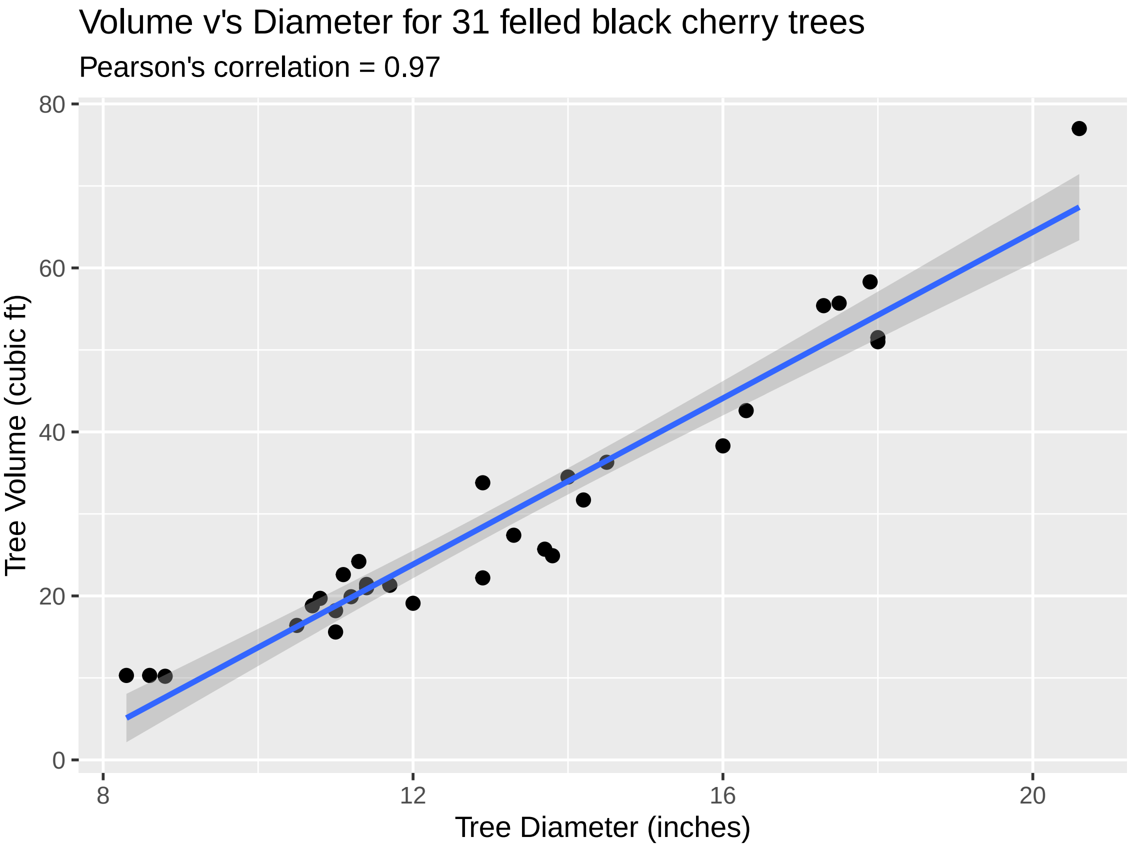
  (Volume = -36.9 + 5.07 x Diameter). We can add the regression line to our scatter plot directly as follows:

```
ggplot(data = trees, aes(x = Diameter, y = Volume)) +
  geom_point(size = 2) +
  labs(x = "Tree Diameter (inches)", y = "Tree Volume (cubic ft)",
       title = "Volume v's Diameter for 31 felled black cherry trees",
       subtitle = paste0("Pearson's correlation = ",round(cor(trees$Diameter,trees$Volume),2))
  geom_abline(slope = coef(trees_lm)[["Diameter"]],
              intercept = coef(trees_lm)[["(Intercept)"]],color = "red")
```



Or, alternatively, we can use the `geom_smooth()` option in `ggplot` to generate and add the linear model fit as a trend line. Here, we must use the argument `method='lm'` to obtain a linear fit (as a default, `geom_smooth()` applies a non-linear model trend, the details of which are beyond the scope of MPS223).

```
ggplot(data = trees, aes(x = Diameter, y = Volume)) +
  geom_point(size = 2) +
  labs(x = "Tree Diameter (inches)", y = "Tree Volume (cubic ft)",
       title = "Volume v's Diameter for 31 felled black cherry trees",
       subtitle = paste0("Pearson's correlation = ",round(cor(trees$Diameter,trees$Volume),2))
  geom_smooth(method='lm')
```



A nice feature of this plotting approach is that `geom_smooth()` automatically includes a 95% confidence interval around the fitted linear regression line. However, we note that the actual details of the line itself are not given by this approach and must be obtained by fitting the simple linear regression model directly, as described above.

12.2.2 Prediction from the fitted model

Our fitted model (the estimated linear regression equation) can be used to predict our dependent variable y (the volume of the tree, in cubic ft) given the value of our independent variable x (the diameter of the tree, in inches, measured at 4ft 6in above the ground).

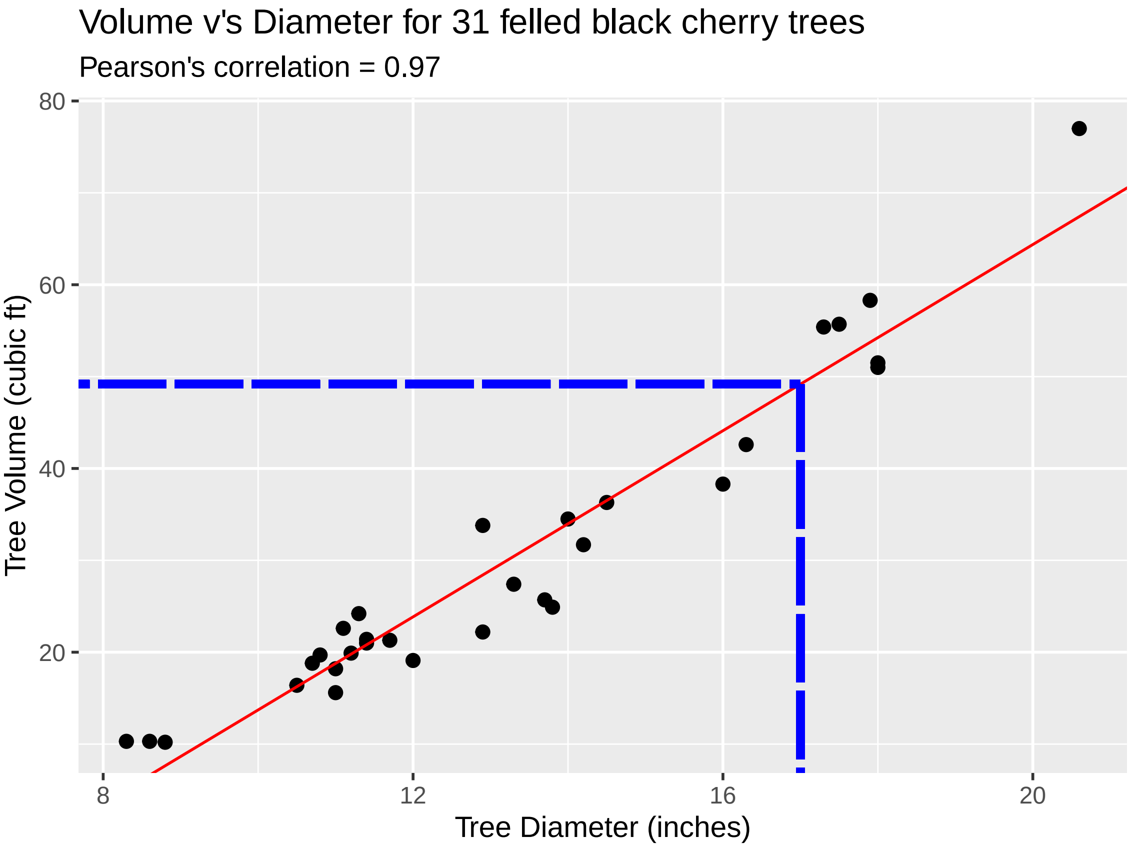
12.2.2.1 Example

If we measure a tree diameter of $x = 17$ inches, what is the predicted volume of this tree from our model?

1. Computing this prediction manually, using the formula:

$$\hat{y} \ = \ -36.9 + 5.07x \ = \ -36.9 + (5.07 \times 17) \ = \ 49.3 \text{ cubic feet (to 1 d.p.).}$$

2. Computing this prediction manually, using the plot:



3. Computing this prediction using R:

```
newdata <- data.frame(Diameter=17)
predict(trees_lm, newdata, se.fit=TRUE)
```

```
## $fit
##       1
## 49.1761
##
## $se.fit
## [1] 1.201876
##
## $df
## [1] 29
##
## $residual.scale
## [1] 4.251988
```

The standard error on this predicted value is very small compared to the predicted value itself. This suggests we have confidence in the accuracy of this prediction.

> When we use a statistical model to make predictions, we must be careful with respect to **extrapolation**. The fitted model only has information about the dependent variable y within the range of the data of the independent variable, x. Outside the range of x, predictions will have much larger uncertainty and may not be valid.
> For example, for the prediction of the tree volume y here, the smallest tree diameter x measured is >8 inches. The intercept of the fitted line is ($x = 0$, $y = -36$), which means that a very small diameter will result in negative volume being estimated. Hence, the model is not valid for very small tree diameters and such extrapolation would yield impossible results.

12.3 Exploring variation over time

When a variable in a data set varies over time it may be described as a **time series** variable and we may be interested in understanding that variation over time better.

A time series is an ordered collection of observations, obtained at successive times and collected in equally spaced time intervals. Time series occur in many application areas, and in particular relevance to our course project, in weather forecasting.

There is a whole branch of statistical modelling and inference techniques that are dedicated to time series analysis - we will explore some key features of a time series here from a simpler EDA perspective, but we will not cover the details of full time series modelling and forecasting - if you are interested in formally modelling time series data, you might want to consider taking the year 3 module: MPS319 - Time Series next year.

In time series data, some **key patterns** in the data that we might explore are:

1. A **trend** in the data - does the data show any long-term pattern? (E.g. increasing, decreasing or no change.)
2. **Seasonality** in the data - are there short-term patterns that occur within a short unit of time and repeat indefinitely? (E.g. a monthly cycle.)
3. **Temporal dependence** in the data - is the observation at time t dependent on the observation at time $t-1$? (Or, time $t-2$, $t-3$, etc.)

Let's consider how we might use EDA to explore these. To illustrate, we will use the data contained in the file `AirPassengers_reformatted.csv`. This data is a reformatted version of the built-in R time series object called `AirPassengers`, which contains monthly totals of international airline passengers for an airline, in thousands, from 1949 to 1960. [See `help(AirPassengers)` for further details and a reference for this data.]

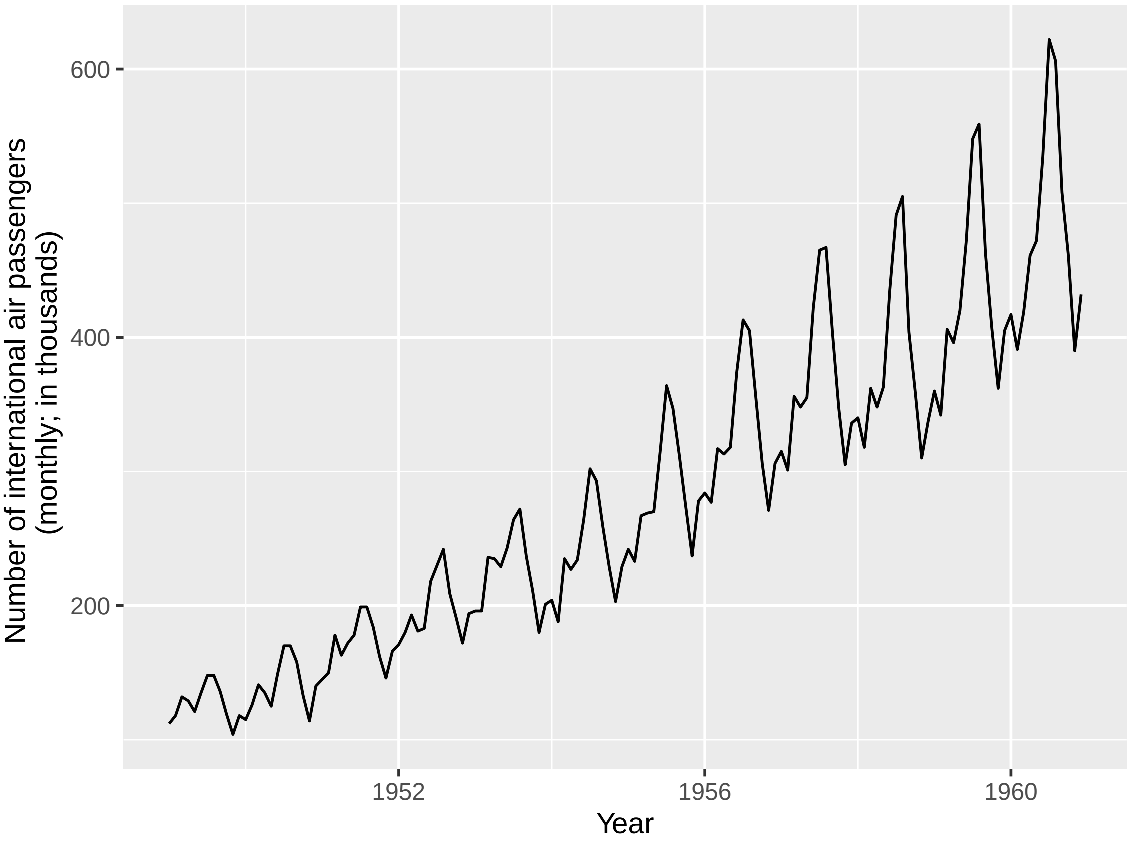
The data is read into R using the following commands (here, we also add a 'decimal time' variable to the data frame to aid in plotting the data):

```
AirPassengerData <- read_csv("data/AirPassengers_reformatted.csv")
Decimal_Year_Vec <- AirPassengerData$year+(AirPassengerData$month/12)-(1/12)
AirPassengerData <- mutate(AirPassengerData,Decimal_Year = Decimal_Year_Vec)
rm(Decimal_Year_Vec)
head(AirPassengerData)
```

```
## # A tibble: 6 x 4
##    year month AirPassengers Decimal_Year
##   <dbl> <dbl>         <dbl>        <dbl>
## 1  1949     1           112        1949
## 2  1949     2           118        1949.
## 3  1949     3           132        1949.
## 4  1949     4           129        1949.
## 5  1949     5           121        1949.
## 6  1949     6           135        1949.
```

and we can visualise the Air Passenger data as a line plot:

```
ggplot(data=AirPassengerData, aes(x = Decimal_Year, y = AirPassengers)) +
  geom_line() +
  labs(x = "Year", y = "Number of international air passengers \n (monthly; in thousands)")
```

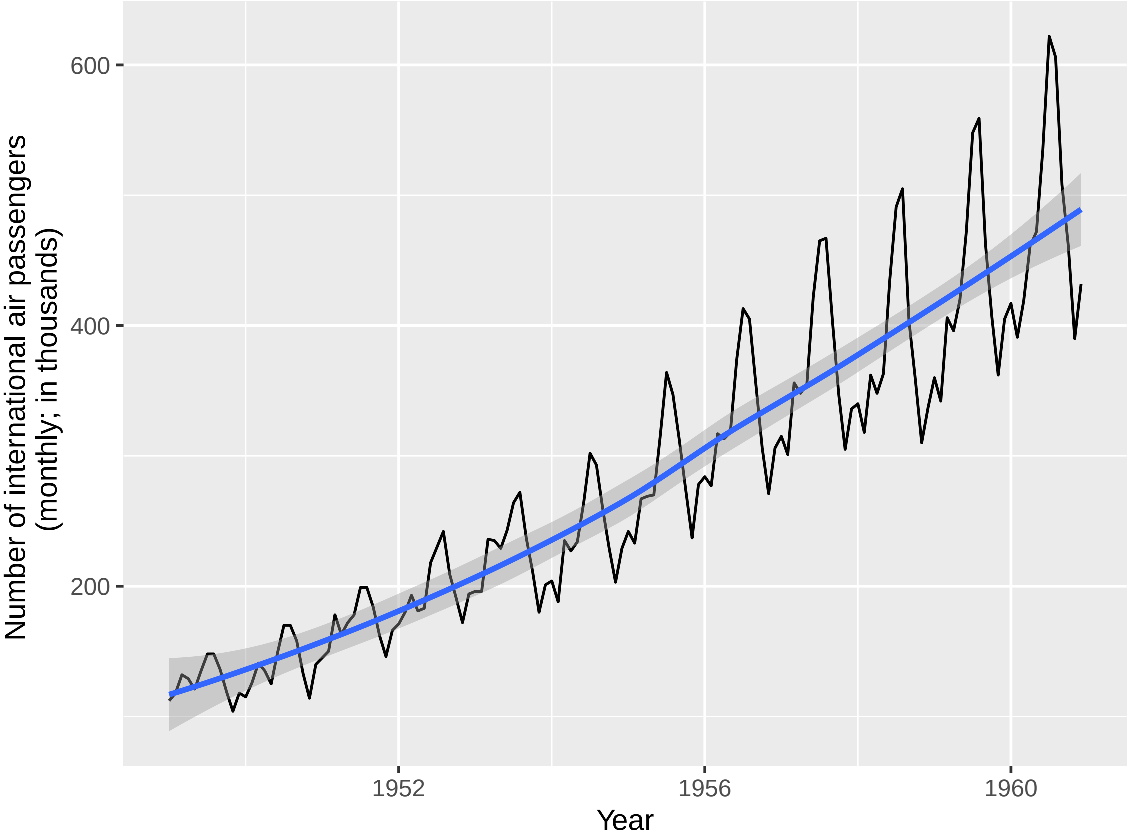


Clearly, this data exhibits a trend, seasonality and potentially has temporal dependence. We will now explore these using EDA methods.

12.3.1 Trends

Although this is not strictly statistically 'valid', as we are not accounting for any dependence/correlation in the time variable (x) here, we can use `ggplot()` to explore the presence of, and comment on, any long-term trends in our data over time by including the `geom_smooth()` argument with our plot. (The default of `geom_smooth()` is to fit a non-linear trend over the data.)

```
ggplot(data=AirPassengerData, aes(x = Decimal_Year, y = AirPassengers)) +
  geom_line() +
  labs(x = "Year", y = "Number of international air passengers \n (monthly; in thousands)") +
  geom_smooth()
```



In this example, there is clearly an increasing trend in the number of international air passengers travelling over time, over the observed period of 1949-1960.

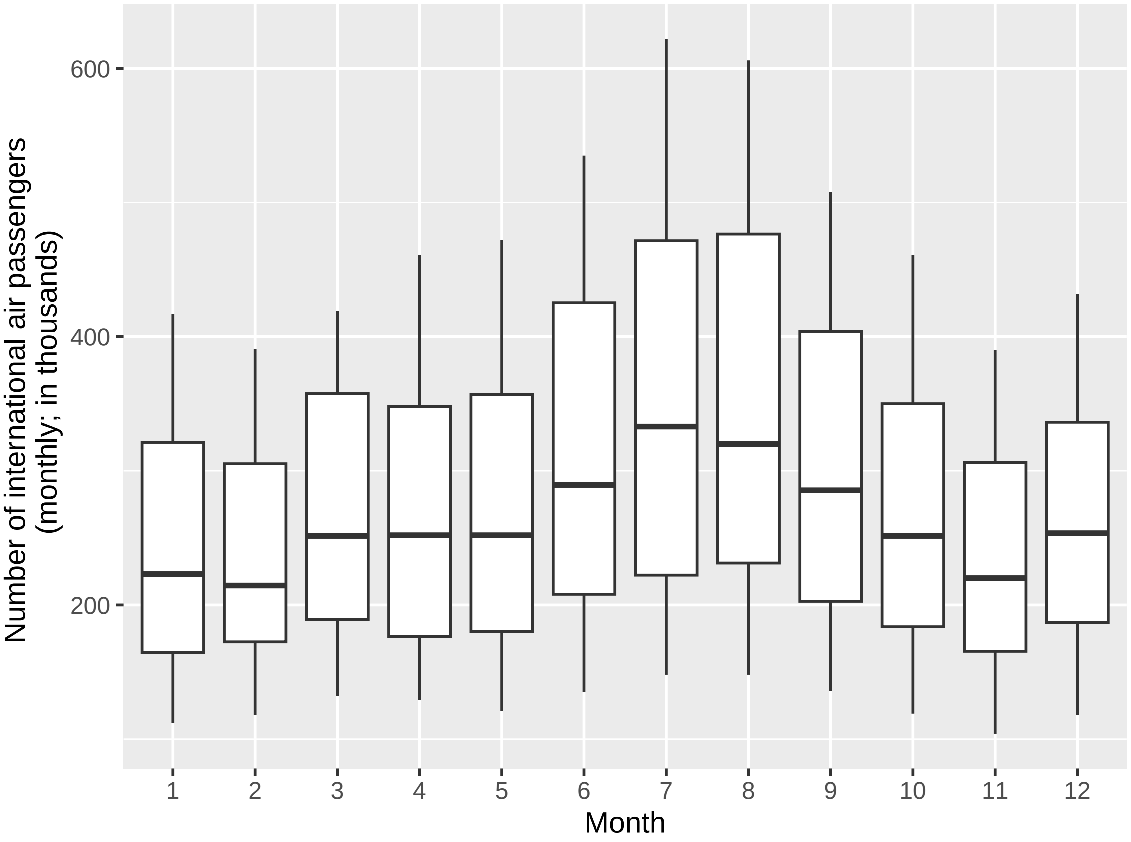
12.3.2 Seasonality

Seasonality in a time series shows as short-term patterns that occur within a short unit of time and repeat over the length of the observed data. Most commonly in environmental data, we see seasonal effects play out across yearly time intervals in the data (i.e. changes/variation in the variable quantity dependent on the season and/or month of the year).

Looking at our line plot of the air passenger data above, we can see that there is clearly a cycle of variation within each year of the data, with lower passenger numbers occurring in winter months and higher passenger numbers occurring over summer months. [Note that the variation in the seasonal pattern increases with time, but that this seems in proportion with the increasing magnitude (trend) of the number of passengers observed.]

To investigate this seasonal behaviour further and quantify the seasonal variation more explicitly, we can break the data down into groups (say individual months or seasons) and apply our EDA techniques over those groupings to evaluate summaries of the patterns. For example, we might produce a figure containing box plots of the air passenger data, grouping the data by the month of the year:

```
ggplot(data = AirPassengerData, aes(x = as.factor(month), y = AirPassengers)) +
  geom_boxplot() +
  labs(x = "Month", y = "Number of international air passengers \n (monthly; in thousands)")
```



This figure shows that the monthly median number of international air passengers (shown by the central line in each box plot) over the period 1949-1960 is higher in summer months like July (`month = 7`) than it is in winter months like January (`month = 1`). The observations in the month of July also have wider variation, and a wider range. [Note: Consider how the above figure could be improved to increase clarity, e.g. the labelling on the x-axis could be much clearer!]

Further / alternative ways to investigate seasonal patterns might be:

- to produce key summary statistics highlighting important differences;
- to compute seasonal average air passenger numbers for each year (using a `for` loop - see the Computing using loops section), and investigate the differences in those;
- to create line plots that show the data for each month as separate lines over the period.

** **Think creatively about how you might visualise the patterns!** **

12.3.3 Temporal dependence

To explore temporal dependence in the data (how an observation at a given time depends on the observations that come directly before it) we will examine how our observed data at time t depends on the observed data at time $(t-1)$. We can calculate an approximate understanding of this as follows.

Firstly, we create a new data frame that contains pairs of observations (y_t, y_{t-1}):

```
N_total <- dim(AirPassengerData)[1]

AirPassengerData_Timelag <- AirPassengerData[-1,-4]
AirPassengers_lag1_Vec <- AirPassengerData$AirPassengers[-N_total]

AirPassengerData_Timelag <- mutate(AirPassengerData_Timelag, AirPassengers_lag1 = AirPassengers
head(AirPassengerData_Timelag)
```

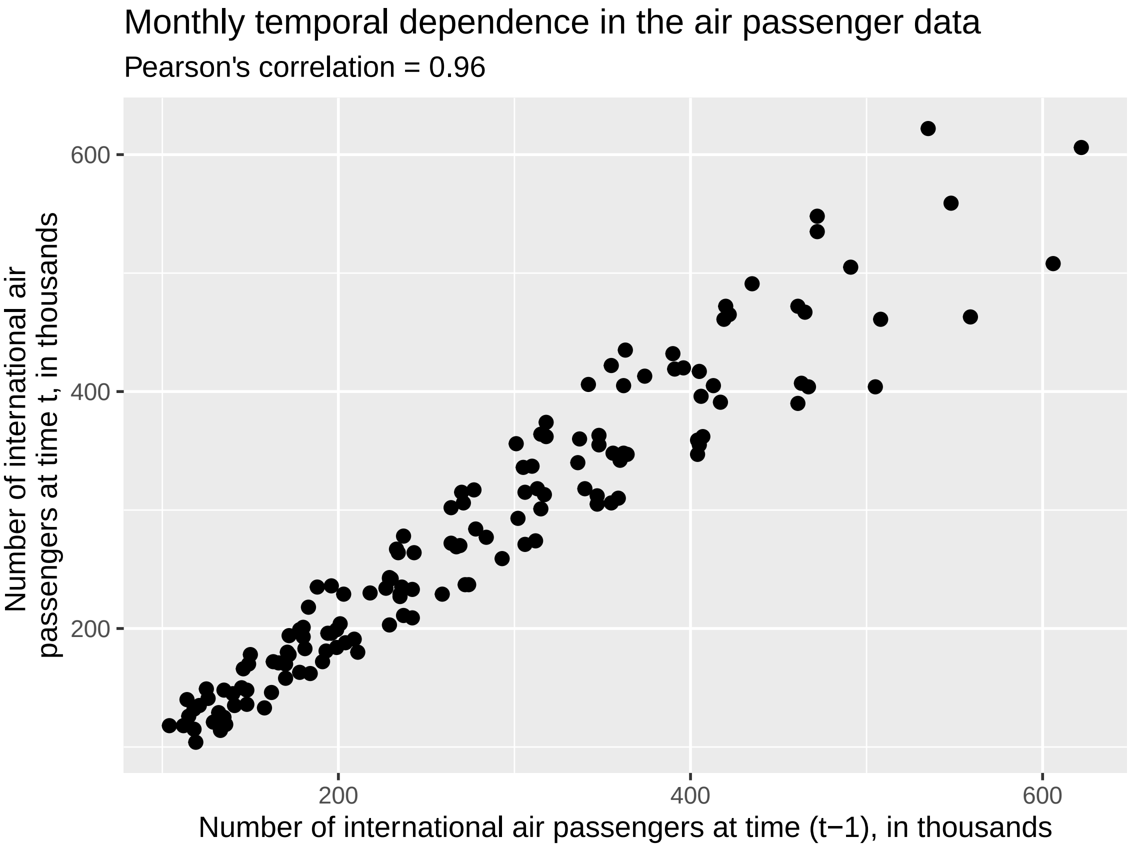
```
## # A tibble: 6 x 4
##    year month AirPassengers AirPassengers_lag1
##   <dbl> <dbl>         <dbl>              <dbl>
## 1  1949     2           118                112
## 2  1949     3           132                118
## 3  1949     4           129                132
## 4  1949     5           121                129
## 5  1949     6           135                121
## 6  1949     7           148                135
```

```
rm(N_total, AirPassengers_lag1_Vec)
```

Here, in this new data frame called `AirPassengerData_Timelag` we have created a new column called `AirPassengers_lag1`, which is the original `AirPassengers` column of the data frame `AirPassengerData`, 'time-lagged' by one observation (1 month). Hence, in the new data frame we can see that we no longer have an observation line for January 1949 - it has been removed as there is no value for this observation for time $(t-1)$ - and, each value in the `AirPassengers_lag1` column (y_{t-1}) appears on the line above in the `AirPassengers` column (as y_t). Data that is time-lagged by one observation in this way (column `AirPassengers_lag1`) is often referred to as a data series **at time lag 1**.

Let's now make a scatter plot of y_t v's y_{t-1} to investigate for any temporal dependence at lag 1 for this data:

```
Temp_Correlation <- cor(AirPassengerData_Timelag$AirPassengers, AirPassengerData_Timelag$AirPas
ggplot(data = AirPassengerData_Timelag, aes(x = AirPassengers_lag1, y = AirPassengers)) +
  geom_point(size = 2) +
  labs(x = "Number of international air passengers at time (t-1), in thousands", y = "Number o
       title = "Monthly temporal dependence in the air passenger data",
       subtitle = paste0("Pearson's correlation = ",round(Temp_Correlation,2)))
```



We can clearly see a strong dependence beetween the observations at time t and the lag 1 observations at time $(t-1)$.

We can use Pearson's correlation coefficient to quantify the strength of this dependence. This calculation is included with the figure above and we obtain a correlation value of $r_{xy} = 0.96$. This indicates a very strong linear relationship for this temporal dependence, which the scatter plot above verifies. This correlation estimate is in fact an approximation for a time series quantity called the **autocorrelation at lag 1**. The linear relationship could also be further quantified using Simple linear regression, if required.

12.4 Exercise

Exercise 12.1. For the monthly mean wind speed measurements (m/sec) in Exercise 11.1 (data file `Monthly_Wind_MA_1980-95.csv`), investigate the variation in the data over time:

1. Make some plots to investigate the presence of any long term trend in the data over time and any seasonal patterns in the data.
2. Explore the temporal dependence in the data by plotting y_t against y_{t-1}.
3. Quantify the strength of this relationship using both Pearson's correlation coefficient and Spearman's correlation coefficient. Describe the relationship you have quantified with words.
4. Use simple linear regression to further quantify a linear relationship between y_t and y_{t-1}. Describe the fitted model. How useful is the wind speed at time $(t-1)$ for describing the wind

speed at time t? How much would you trust a prediction from this simple linear regression model?

Chapter 13

Imputing missing data

It is common to find missing values when provided with a data set. In this Section, we'll briefly consider some simple options for 'imputing' (estimating) missing data, should that be appropriate. There are different options (and R packages) for doing this. We will briefly illustrate one package, `imputeTS` (Moritz and Bartz-Beielstein, 2017). This package has a nice 'cheat sheet' which illustrates its functions.

Suppose we have a vector with some missing values, which we will create as follows, and treat as a time series.

```
set.seed(123)
x <- signif(1:10 + rnorm(10), 3)
x[c(3, 4, 8)] <- NA
print(x)
```

```
##  [1] 0.44 1.77   NA   NA 5.13 7.72 7.46   NA 8.31 9.55
```

Then, some options for imputing the missing values are:

- impute using the mean of all the non-missing cases

```
imputeTS::na_mean(x)
```

```
##  [1] 0.440000 1.770000 5.768571 5.768571 5.130000 7.720000 7.460000 5.768571
##  [9] 8.310000 9.550000
```

- impute using the most recent observed value, "last observation carried forward" (e.g. estimate `x[3]` by `x[2]`)

```
imputeTS::na_locf(x)
```

```
##  [1] 0.44 1.77 1.77 1.77 5.13 7.72 7.46 7.46 8.31 9.55
```

- impute using linear interpolation (e.g. linearly interpolate between `x[2]` and `x[5]` to get `x[3]` and `x[4]`, assuming the observations are uniformly separated in time)

```
imputeTS::na_interpolation(x)
```

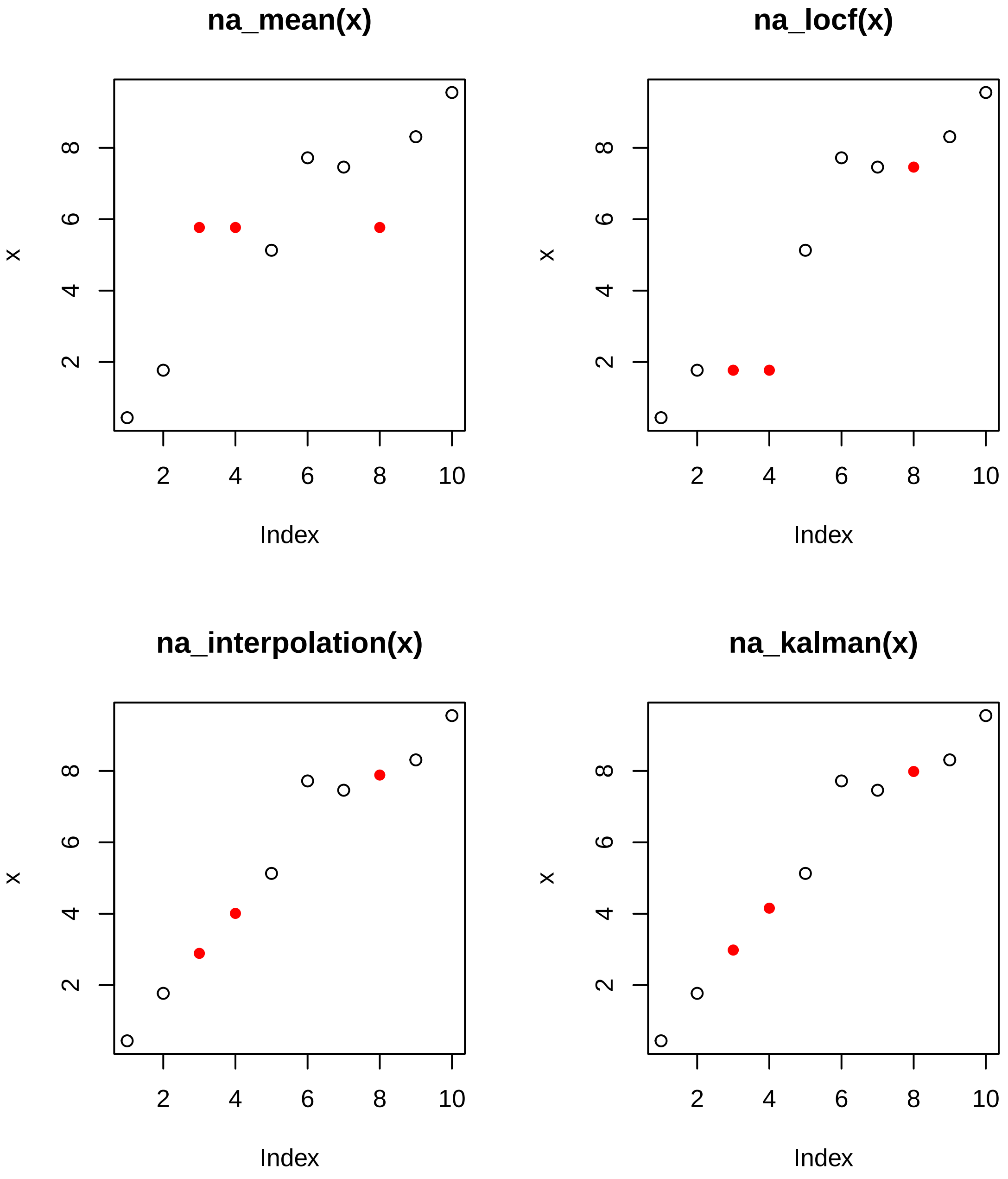
```
##  [1] 0.440 1.770 2.890 4.010 5.130 7.720 7.460 7.885 8.310 9.550
```

- impute using a Kalman smoother (a time series approach)

```
imputeTS::na_kalman(x)
```

```
##  [1] 0.440000 1.770000 2.982747 4.155760 5.130000 7.720000 7.460000 7.985059
##  [9] 8.310000 9.550000
```

The plot below shows the imputed values (as red circles) in each case.



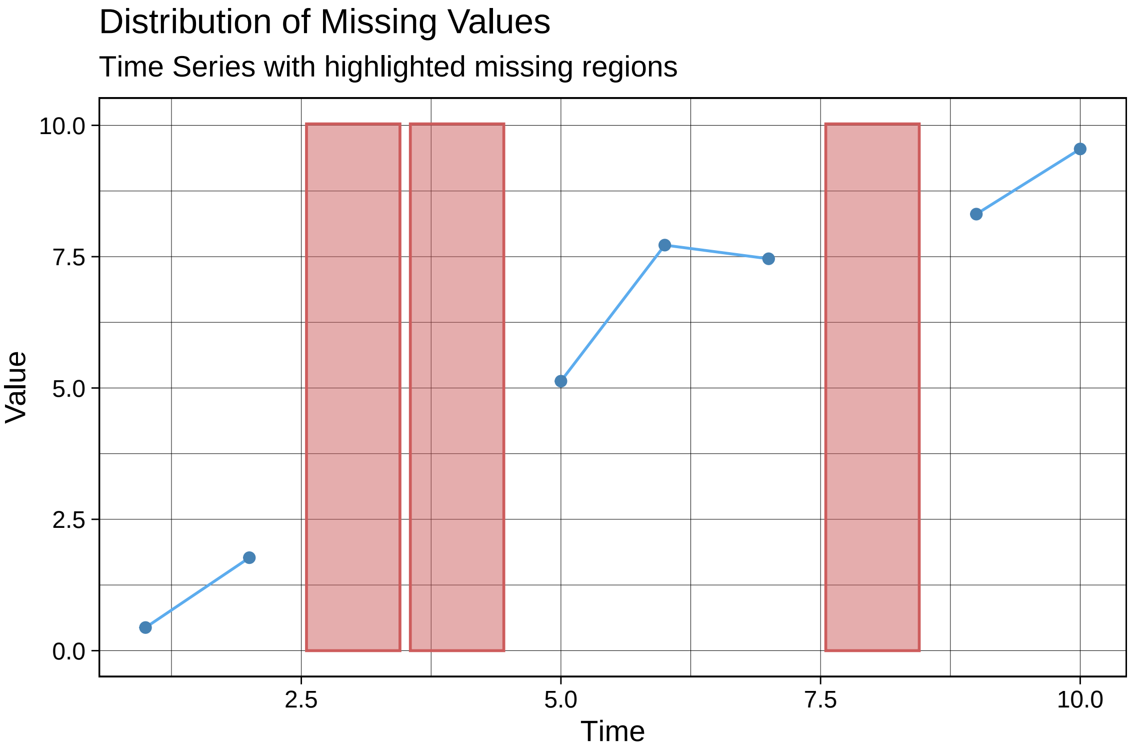
Modelling-based estimates such as those from the Kalman smoother typically involve models in which we observe some process of interest *plus noise/measurement error*. Estimates obtained from imputation would be of this 'underlying' process, not estimates of what would actually be *observed*.

13.1 Visualising missing data

The imputeTS package has some nice plotting functions for missing data. The plots make use of ggplot2 (which we covered earlier), but you don't need to know any additional ggplot2 syntax to use these functions.

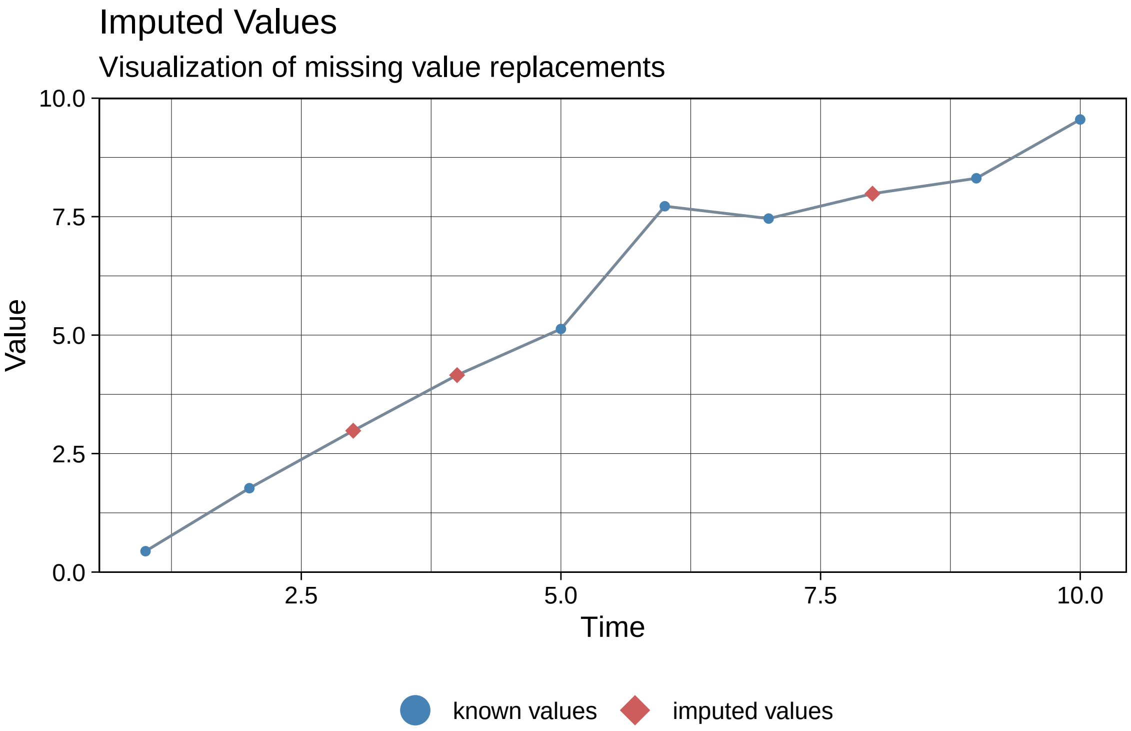
We can produce a plot to show clearly where the missing data are using

```
imputeTS::ggplot_na_distribution(x)
```



and if we have used imputation, we can make a plot that clearly displays the imputed values with (using na_kalman() as an example):

```
imputeTS::ggplot_na_imputations(x,
                                imputeTS::na_kalman(x))
```



13.2 Exercise

Exercise 13.1. The built-in data frame `airquality` includes time series data of four variables, and has missing values in the `Ozone` and `Solar.R` variables.

Use the command `data(airquality)` to load this data into your R environment and produce plots that:

1. indicate where these missing observations are;
2. show imputed values using the last observed observation for each variable.

Bibliography

Auguie, B. (2017). *gridExtra: Miscellaneous Functions for "Grid" Graphics.* R package version 2.3.

Cheng, J., Karambelkar, B., and Xie, Y. (2019). *leaflet: Create Interactive Web Maps with the JavaScript 'Leaflet' Library.* R package version 2.0.3.

Moritz, S. and Bartz-Beielstein, T. (2017). imputeTS: Time Series Missing Value Imputation in R. *The R Journal*, 9(1):207–218.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York.

Wickham, H. (2019). *stringr: Simple, Consistent Wrappers for Common String Operations.* R package version 1.4.0.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2020a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics.* R package version 3.3.2.

Wickham, H., François, R., Henry, L., and Müller, K. (2020b). *dplyr: A Grammar of Data Manipulation.* R package version 1.0.0.

Wickham, H., Hester, J., and Francois, R. (2018). *readr: Read Rectangular Text Data.* R package version 1.3.1.